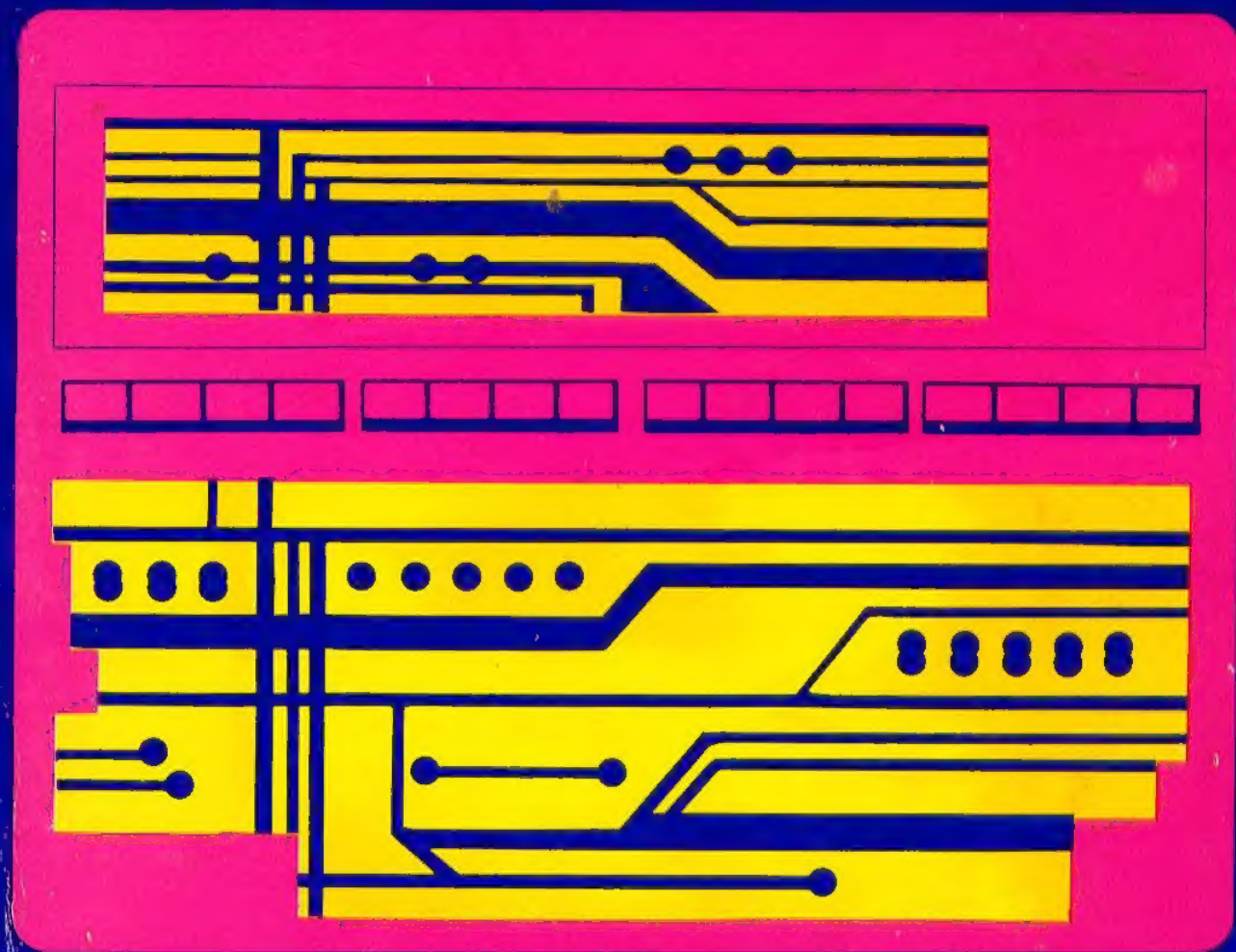


# Inside the Model 100



8085 Assembly Language Programming  
Advanced BASIC programming  
Hardware Overview-Keyboard, LCD, Printer Interface,  
Cassette I/O, Beeper, Power Supply, RS-232, Modem

**Carl Oppedahl**

---

Inside The  
TRS-80® Model 100

---



---

Inside The  
TRS-80® Model 100

---

by  
**Carl Oppedahl**

Weber Systems, Inc.  
Cleveland, Ohio

The authors have exercised due care in the preparation of this book and the programs contained in it. The authors and the publisher make no warranties either express or implied with regard to the information and programs contained in this book. In no event shall the authors or publisher be liable for incidental or consequential damages arising out of the furnishing, performance, or use of this book and/or its programs.

Touch-Tone® is a registered trademark of AT&T; Epson®, MX-80™ is a registered trademark of Epson Corporation; Polaroid is a trademark of Polaroid Corporation; TRS-80® Model 100 is trademark of Radio Shack division of Tandy Corporation; Teletype™ is a trademark of Teletype Corporation; Telex™ is a trademark of Telex Communications, Inc.; Z80™ is a trademark of Zilog Corporation.

Published by:  
Weber Systems, Inc.  
8437 Mayfield Road  
Chesterland, Ohio 44026

For information on translations and book distributors outside of the United States, please contact WSI at the above address.

**Inside the TRS-80® Model 100**

Copyright© 1985 by Weber Systems, Inc. All rights reserved under International and Pan-American Copyright Conventions. Printed in United States of America. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopy, recording, or otherwise without the prior written permission of the publisher.

**Library of Congress Cataloging in Publication**

Main entry under title:

Inside the TRS-80® Model 100

Includes index

1. TRS-80 Model 100 (Computer) I. Title.

QA76.8.T184566 1985 001.64 85-11560

ISBN 0-938862-31-6

---

# Contents

---

<b>Preface</b>	<b>17</b>
<b>1. Introduction to The Model 100</b>	<b>21</b>
Why Machine Language Programming?	21
What is Machine Language?	22
The CPU	24
Hexadecimal Notation	25
The Opcode Instruction Set	26
Assemblers	26
<b>2. Assembly Programming</b>	<b>29</b>
Architecture	29
The Accumulator and the Flag Register	31
The Other Registers	32
Clock Frequency and Execution Speed	33
The Opcodes	33
Memory Moves	36
Other Eight-bit Moves	37
Sixteen-bit Data Transfers	38
Stack Operations	39
Branch Instructions	40
Subroutine Calls	41
Conditional Calls and Returns	42
Restart Instructions	43
Another Jump Instruction	45

Arithmetic Functions .....	46
Addition with Carry .....	47
Subtraction .....	48
Eight-bit Increments and Decrements .....	49
Binary-Code Decimal Operations .....	49
Sixteen-bit Arithmetic .....	50
Logical Operations .....	51
Turning a Bit On .....	52
Turning a Bit Off .....	53
Testing a Register Pair for Zero .....	53
Comparison Operations .....	53
Rotate Instructions .....	54
Uses for the Rotate Instructions .....	56
Other Logical Instructions .....	56
The Carry Flag .....	57
Machine Control Instructions .....	57
<b>3. Advanced BASIC .....</b>	<b>59</b>
PEEK .....	60
POKE .....	61
Input Ports .....	62
Allocating RAM-The Maxram and Himem Values .....	63
Himem .....	64
SAVEM,CSAVEM .....	66
LOADM AND CLOADM .....	67
VARPTR .....	68
CALL .....	68
Menu Selections of CO Files .....	69
Battle of the CO Files .....	70
Poking Into Protected Memory .....	70
<b>4. Borrowing From Z80 Experience .....</b>	<b>73</b>
Differences Between the Z80 and 80C85 Instruction Sets .....	73
<b>5. Understanding the Hardware of the Model 100 .....</b>	<b>79</b>
Memory Locations .....	80
The Ports .....	86
Connections in the Model 100 .....	89
<b>6. The Keyboard .....</b>	<b>93</b>
Hardware Theory of Operation .....	93
Keyboard Scanning .....	96
Multiple-Use Ports .....	99
Keyboard ROM Calls .....	100

Keyboard Input ROM Routines	101
Label Lines and Function Keys	103
ROM Keyboard Scanning	105
Decoding of Function Keys	106
Decoding the Directional Arrows	106
Nums Decoding	108
Arriving At A Particular INKEY\$ Value	108
<b>7. UART Operation and the RS-232 Interface</b>	<b>117</b>
Parallel and Serial Data	117
Converting From Parallel Data To Serial Data	118
What is a UART?	118
Timing	120
The Inner Workings of the UART	121
CPU Communication with the UART	123
Serial WORD Configuration	123
Setting the BAUD Rate	125
Serial Transmission	126
Serial Data Reception	126
Data Reception Errors	129
Significance of ASCII Code	130
The Beeper	132
ASCII Protocol — XON/XOFF	132
Mode Selection:RS-232 and Modem	132
The RS-232 Standard	135
Mechanical Requirements	135
Electrical Requirements	135
Data Flow	137
Handshaking Signals	137
How The RS-232 Interface Works	138
Receiving RS-232 Data	138
Transmitting RS-232 Data	140
Published ROM Subroutines	140
<b>8. The Telephone Modem</b>	<b>143</b>
Data Flow Overview	143
The Bell 103 Standard	144
The Audio Frequencies	147
How Telephones Work	148
The Telephone Wires	148
Electrical Considerations	148
Placing Telephone Calls	148
Answering Telephone Calls	149
Ringer Equivalence	149



The Direct-Connect Modem And The Telephone Transformer OT1	149
Ring Pause	152
FCC Certification	152
Modem Data Flow	153
The Modem Receiver	153
Outgoing Data Path	157
Acoustically-Coupled Modem	157
Use Of The Coupler	159
Dialing Procedures With The Coupler	159
I/O Ports	161
ROM Subroutines	161
<b>9. Piezoelectric Beeper</b>	<b>163</b>
How Piezo Beepers Work	163
Hardware Theory of Operation	166
CPU Toggling	168
PIO Timer Use	171
Musical Tones	172
<b>10. The Printer Interface</b>	<b>175</b>
Mechanical Requirements	175
Electrical Requirements	177
Software Characteristics	177
Model 100 Printer Hardware	178
How the ROM Prints Characters	179
Fancier Print Routines	182
ROM Calls to the Printer	182
PRINTR	182
PRTLCD	183
PRTTAB	183
Printing to Dot-Addressable Graphic Printers	183
Unpublished ROM Routines	183
The Low Battery Light	184
<b>11. Clock/Calendar</b>	<b>187</b>
Terminology	187
Hardware Theory of Operation	188
Setting The Time In The Clock/Calendar	190
Strobing The Clock/Calendar	191
Reading The Time	193
Selecting a TP Frequency	193
Clock/Calendar Accuracy	198
Published ROM Routines	198
Unpublished Routines	199
Setting The Time Through ROM Calls	199

<b>12. Cassette Input and Output</b>	<b>201</b>
Accessing Data (DO) Tape Files From BASIC	202
Creating a CO Cassette File	202
Loading a CO File Back into RAM	202
Accessing BA Tape Files From BASIC	203
Hardware Theory of Cassette Operation	203
Incoming Cassette Data Flow	204
Reading the Data at the SID Terminal	206
Outgoing Cassette Data Flow	207
The CPU's Role	207
Hardware Treatment of the SOD Signal	207
Interrupts	209
Motor Control	209
Published ROM Subroutine Calls	210
Writing to Cassette	210
Reading from Cassette	211
Unpublished Routines	211
File Formats	212
User Experimentation	212
<b>13. The Liquid-Crystal Display Screen</b>	<b>215</b>
How Liquid Crystals Work	215
CPU Control of the Screen	217
Character Formation	217
Formation of Character Shapes	218
RAM Locations Relating to the Display	223
Published ROM Subroutine Calls	223
How To Send Special Characters	226
Sending Characters to the Printer	227
How to Call 4B44	227
Other Published LCD ROM Routines	227
Cursor Position Routines	228
Unpublished ROM Routines for the LCD	229
<b>14. The Bar Code Reader</b>	<b>231</b>
Determining When To Start Reading a Bar Code	232
Handling The Interrupt	234
Polling The Bar Code Reader	234
Reading The Bar Code	235
Using The BCR Connector for Purposes Other Than Reading Bar Codes	236

<b>15. Interrupts</b>	<b>239</b>
Interrupt Priorities	242
Masking and Disabling of Interrupts	242
Uses for the RIM Instruction	244
<b>16. The Power Supply</b>	<b>247</b>
The DC-to-DC Converter	249
Memory Power	249
Low-Power Signals	252
Reset Circuitry	253
Powering Up The CPU	254
The AC Adapter	254
Alternative Power Supply	256
<b>17. Expansions</b>	<b>257</b>
The Bar-Code Reader and CRT	257
Unused Pins	259
Disk Input/Output	259
ROM Routines For Bulk Data Transfer	259
Record I/O	259
Listing Files To The Printer	260
Function Keys in Telcom Term Mode	260
Understanding the Option ROM Socket	260
Expansion Bus Socket	262
Memory Access At The Expansion Connector	263
Address Decoding	264
Adding Ports to the Model 100	264
Parallel Port Input	264
Parallel Port Output	266
Interrupts at the Connector	266
The Telephone Ring Pulse Input	266
Theory of Operation	266
<b>18. The Remainder of ROM</b>	<b>271</b>
Published ROM Initialization Routines	271
Published RAM File-Handling Routines	272
Suzuki and Hayash	273
DO Files	273
BA File Format	273
Block Moves	276
Lowercase Conversion	276
Converting Numerical Hex to ASCII	276
Converting Two Numerical Hex Bytes to ASCII	277
Register-Pair Comparison	277
Utility For Command Decoding	277
RAM Variable Map	277

<b>Appendix A. Nonprintable Characters and Assignments</b>	<b>281</b>
<b>Appendix B. ROM Map</b>	<b>283</b>
<b>Appendix C. 8080, 8085, Z80 Opcodes</b>	<b>295</b>
<b>Appendix D. Bibliography</b>	<b>321</b>
<b>Memory Index</b>	<b>323</b>
<b>Alphabetical Index</b>	<b>327</b>



---

## Preface

---

### **How To Use This Book**

This book is intended for several readers: those who wish to do machine language programming; those who wish to do sophisticated BASIC programming; and those who wish simply to understand how the Model 100 works inside.

### **Understanding the Model 100**

Some Model 100 owners are reasonably content using existing software, and so do not really have programming in mind. For these readers, it is my hope that this book will do two things: provide an explanation of how the computer “really works inside”; and perhaps arouse a little curiosity about the interesting world of machine language programming. It is possible to read the entire book from front to back without assembling a single opcode, or typing `RUNM` even once. Along the way you may still find yourself asking “I wonder how they do that?” If even once you can say, “Ah, now I see how it is done,” then this book will have fulfilled its purpose.

## **Doing Machine Language Programming**

For those who wish to do machine language programming, the course of study depends on your previous experience with machine language.

If you have never programmed in machine language before, you should first become reasonably familiar with Model 100 BASIC, including such statements as OPEN, LINE INPUT#1, PRINT#1. You should also learn to use TEXT, the Model 100 word-processing program.

Then read this book starting from the very beginning, skipping only chapter 4. If you feel comfortable with the subjects covered up to and including chapter 5, then proceed with the rest of the book. You will be able to do almost anything you want with the Model 100.

You may find, after reading those chapters, that you need a more general introduction to microprocessors; a list of suggested books appears in the appendices.

If you have programmed in machine language, whether on another microprocessor or on a large computer, you will be able to start right in with chapters 2 and 3, which introduce the 8085 CPU used in the Model 100. Proceed to chapter 5 which describes the hardware and from there read to the end of the book.

If your previous experience is with one or more processors in the 8080 family, you should have no difficulty starting right in with chapter 5. Those whose first computer was a TRS-80 Model I, III, or IV, and who thus know about the Z80 will want to read chapter 4 closely, as it tells how to use Z80 experience and programming aids on the Model 100.

## **Advanced BASIC Programming**

Some readers have mastered everything in the Model 100 owner's manual, including the complete BASIC command vocabulary. Many want to do more, but do not want to learn to do machine programming. The coverage in the owner's manual is sparse, at best, regarding such commands as PEEK, POKE, INP, and OUT, CALL, RUNM, and the like. This book explains the internal features of the Model 100 that may be accessed and controlled through these commands.

If your BASIC programming goal relates to a particular part of the computer, say, the clock calendar integrated circuit, you may go directly to the chapter on that subject. This chapter describes ways to use the CALL, INP, OUT, PEEK, and POKE commands to perform task, which cannot be accomplished using the common BASIC commands.

For a survey of the many hardware areas which may be controlled with INP and OUT commands, read chapter 5.

Corrections and suggestions for improvement are welcomed, and should be sent to the author.

### **About The Author**

Carl Oppedahl, a lawyer specializing in technological litigation, is a regular contributor to Portable 100 200 and PCM magazines. He holds a bachelor's degree in physics and mathematics from Grinnell College and a law degree from Harvard University. He is an associate with the New York law firm of Kreindler & Kreindler.

### **Acknowledgements**

The author would like to thank Alan Buck, whose unselfish efforts brought the world of computer programming to hundreds of high school students a decade before it was fashionable to do so; Harold Liljedahl, who taught me that when reassembling equipment I should put all the screws in part way before tightening them; Gerald Robbie, who taught me most of what I know about writing; and Lee Kreindler, whose encouragement and support suffice to make almost anything possible.







---

## Introduction

---

### **Why Machine Language Programming?**

The Microsoft BASIC and the nice TEXT and TELCOM programs provided with the Model 100 allow you to undertake an incredible range of programming tasks. Custom software from other suppliers allows one to do almost anything, without having to learn machine language or study opcodes.

Why, then, learn machine language? Machine language is fast—often ten times faster than an equivalent BASIC program, and more compact. If TEXT and TELCOM had been written in BASIC rather than machine language, they would have taken up at least three times the 6.5K they presently occupy in ROM.

The TEXT program, if written in BASIC, would require four minutes for a simple “Find” search in a large file; in machine language it takes less than fifteen seconds.

The TELCOM program simply had to be written in machine language; when characters are arriving at the rate of thirty or more per second and a download to RAM and printer echo are occurring, BASIC simply cannot keep up.

Fast-action games are painfully slow in BASIC; exciting graphics and sound are possible only in machine language.

These are some of the reasons people learn to program in machine language.

### **What Is Machine Language?**

Machine language is just what it sounds like, the language used within the machine. Actually computers are always executing machine language “deep down inside”, even when we think of them as executing BASIC, or FORTRAN, or whatever.

The Model 100 computer contains a microprocessor, sometimes called a CPU (for central processor unit), and a variety of memory and I/O (input and output) devices, as shown in figure 1.1.

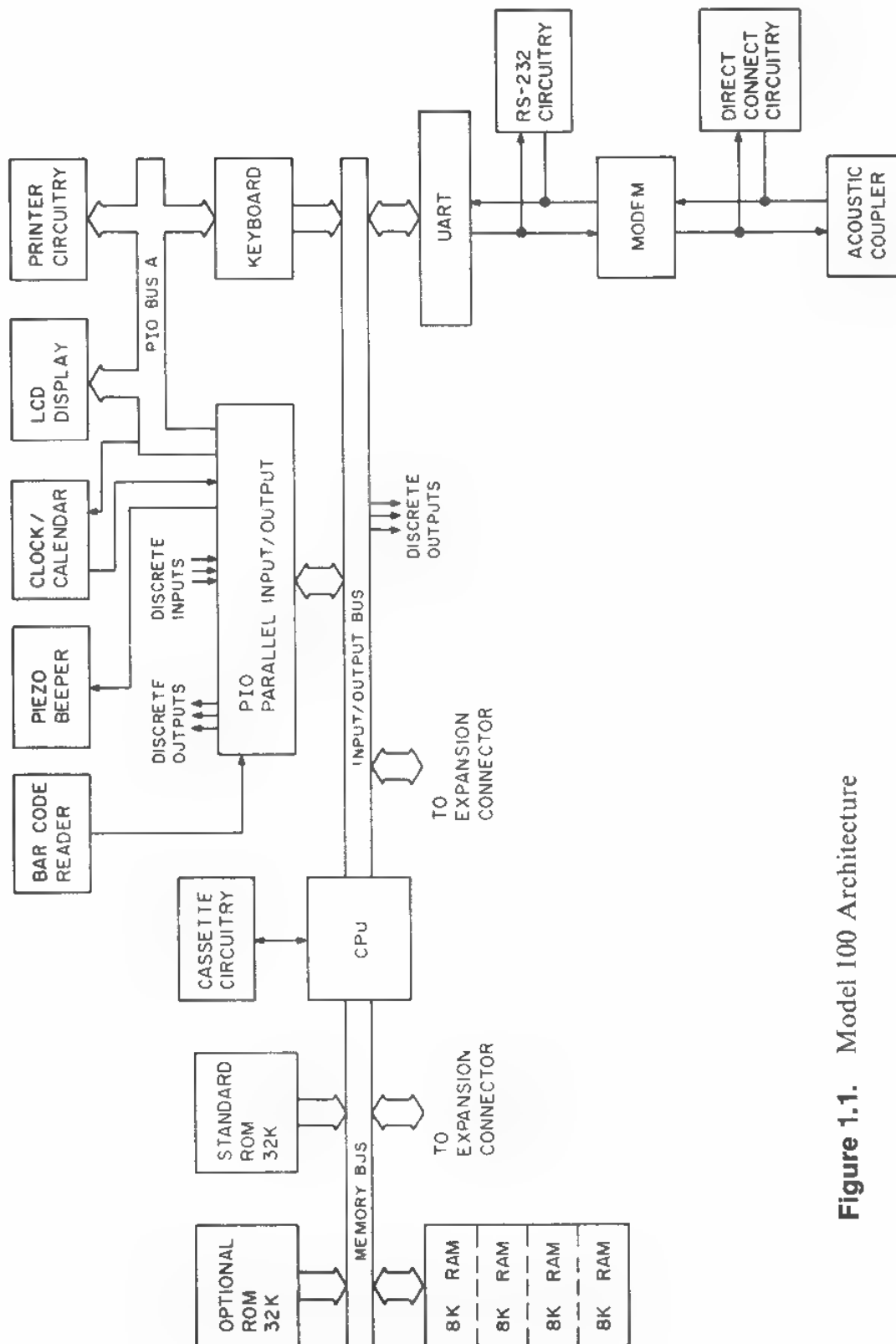


Figure 1.1. Model 100 Architecture

The CPU can communicate directly with the cassette connector; can use the memory bus to store and retrieve information; and can use the input/output (I/O) bus to control and exchange information with a variety of devices, including the keyboard, the liquid-crystal display (LCD) screen, a printer, the telephone line, or any RS-232 device.

That is all fine and good, but how does it really work? For example, when the power is turned on, what exactly happens to result in the time, date and file menu being displayed? When one moves the cursor to one of the file names, and pushes ENTER, what exactly happens?

The answer to such questions lie in an understanding of the way the CPU deals with the circuitry around it, and in a knowledge of the ROM operating system.

The inner workings of the CPU and the operating system provide a background that is presently throughout this book, regardless of the device named in the title of particular chapter.

## **The CPU**

The CPU executes instructions called "opcodes", which are made available to the CPU in memory locations. When power is turned on, the CPU goes to the opcode located at memory location 0000. As it happens, the designers of the Model 100 set up the hardware so that memory location 0000 is in read-only memory (ROM).

Memory location 0000 contains the hexadecimal value C3, or 195 decimal. (You can see this by typing `PRINT PEEK (0)` while in BASIC.)

The 8085 CPU is designed so that when it encounters the value C3, it prepares to "jump", which means that instead of proceeding with the opcode after C3, it proceeds with an opcode elsewhere in memory. The next two storage locations contain information used by the CPU to figure out where it will jump to. (The jump instruction is analogous to the GOTO command in BASIC.)

The next two locations contain 33 hexadecimal (51 decimal) and 7D hexadecimal (125 decimal). The CPU takes the 33 and 7D groups them as 7D33, and jumps to 7D33.

## Hexadecimal Notation

This is as good a time as any to introduce hexadecimal notation. Most addresses, port numbers, and register contents (all defined later) will be given in hexadecimal, usually called “hex”, notation. The reason is that everything in the Model 100 (and indeed everything in every microcomputer) is represented by 1’s and 0’s usually in groups of eight called “bytes”.

Each byte is capable of assuming 256 different values, since there are 256 possible combinations of “1” and “0”. Sometimes the value of a particular byte is conveyed with a decimal number from 0 to 255, using a convention assigning decimal values to the individual 1’s and 0’s, which are called “bits”.

For example, the eight-bit value 01110001 has a decimal value of 113, arrived at in this fashion:

$$(0 \times 128) + (1 \times 64) + (1 \times 32) + (1 \times 16) + (0 \times 8) + (0 \times 4) + (0 \times 2) + (1 \times 1)$$

The numbers 128, 64, and so on are powers of two: 128 is two raised to the seventh power, 64 is two raised to the sixth power, and so on. The “0” bit which was multiplied with 128 is thus often referred to as “bit 7”, the “1” which was multiplied with 64 is called “bit 6”, and so on.

This describes the relation between binary numbers and decimal (base ten) numbers. The connection with hexadecimal is as follows: a group of four bits is matched up with a letter or numerical digit, so that a group of eight bits is matched with a two-letter (or two-digit) combination.

The four-bit values are:

Binary	Hex
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Using this correspondence, the binary value 01110001 translates to 71 hex.

## **The Opcode Instruction Set**

As mentioned above, the 8085 executes opcodes. Thus far, only one opcode has been mentioned, the C3 opcode which means "jump". By the end of chapter 2, the other opcodes which the 8085 is willing to execute will have been discussed. These include instructions which cause information to be moved from one place to another (the equivalent of LET A=B); which will test for presence of certain conditions and then jump (the equivalent of IF statements); and which add or subtract integers.

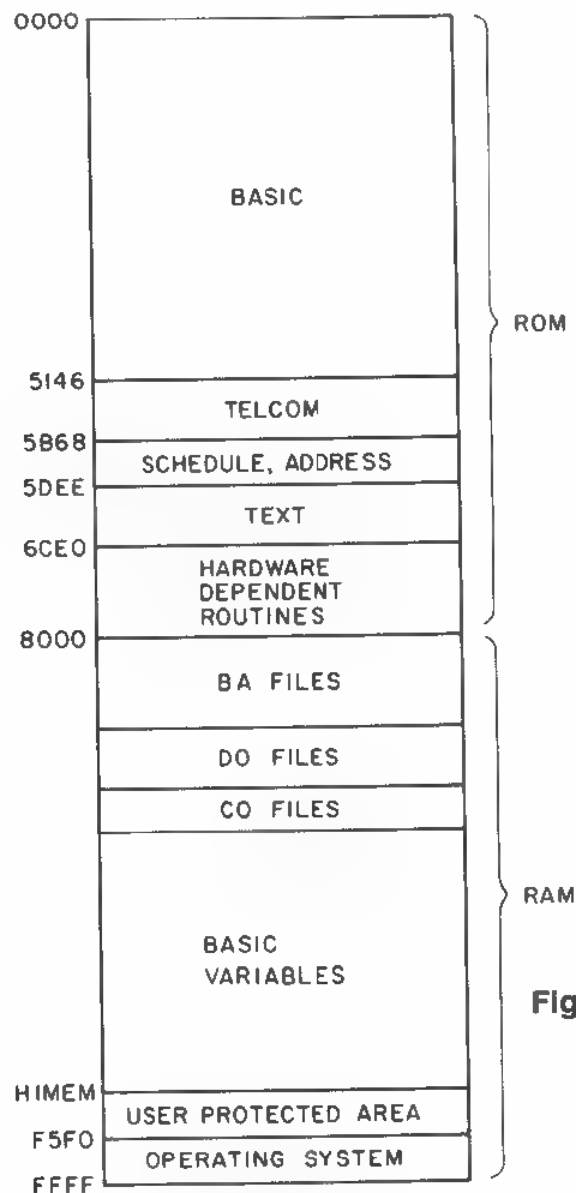
## **Assemblers**

Each opcode is an eight-bit byte, and in some cases the one or two bytes following the opcode also form part of an instruction. The two

bytes following the "C3" discussed above give an example of the latter. Thus an instruction may be one, two, or three bytes long.

The bottom 32,768 (decimal) Model 100 memory locations, which are in ROM (running from 0000 to 7FFF), together with the top 2574 (decimal) memory locations, which are in RAM (running from F5F0 to FFFF), are filled with opcodes and related data placed there by Microsoft. Those opcodes, which may be grouped into subroutines and programs, make possible the menu that will be displayed when you turn on the machine as well as the events which occur when you run the applications programs.

The allocation of memory is shown to scale in figure 1.2.



**Figure 1.2.** Memory Map



When Microsoft developed the opcodes, they were doing machine language programming. You may be sure they did not start by creating a list of hex opcodes and running these. Instead, they followed a development process involving flowcharts, mnemonics, and a variety of programming aids such as assemblers, debuggers and disassemblers.

# 2

---

## Assembly Programming

---

### Architecture

The 8085 CPU is composed of several parts, as shown in figure 2.1. Signals enter and exit from the CPU at the locations shown in the figure. Sixteen wires are available for selecting which of the 65536 memory addresses is being referred to. Eight of the wires are also used for incoming and outgoing data. The signals shown across the top of the figure have to do with serial input and output, and interrupts, discussed in detail in chapters 12 and 15, respectively. Timing and control signals shown at the bottom of the figure, let the CPU determine whether input or output will occur, and whether the input or output will take place in port space or address space.

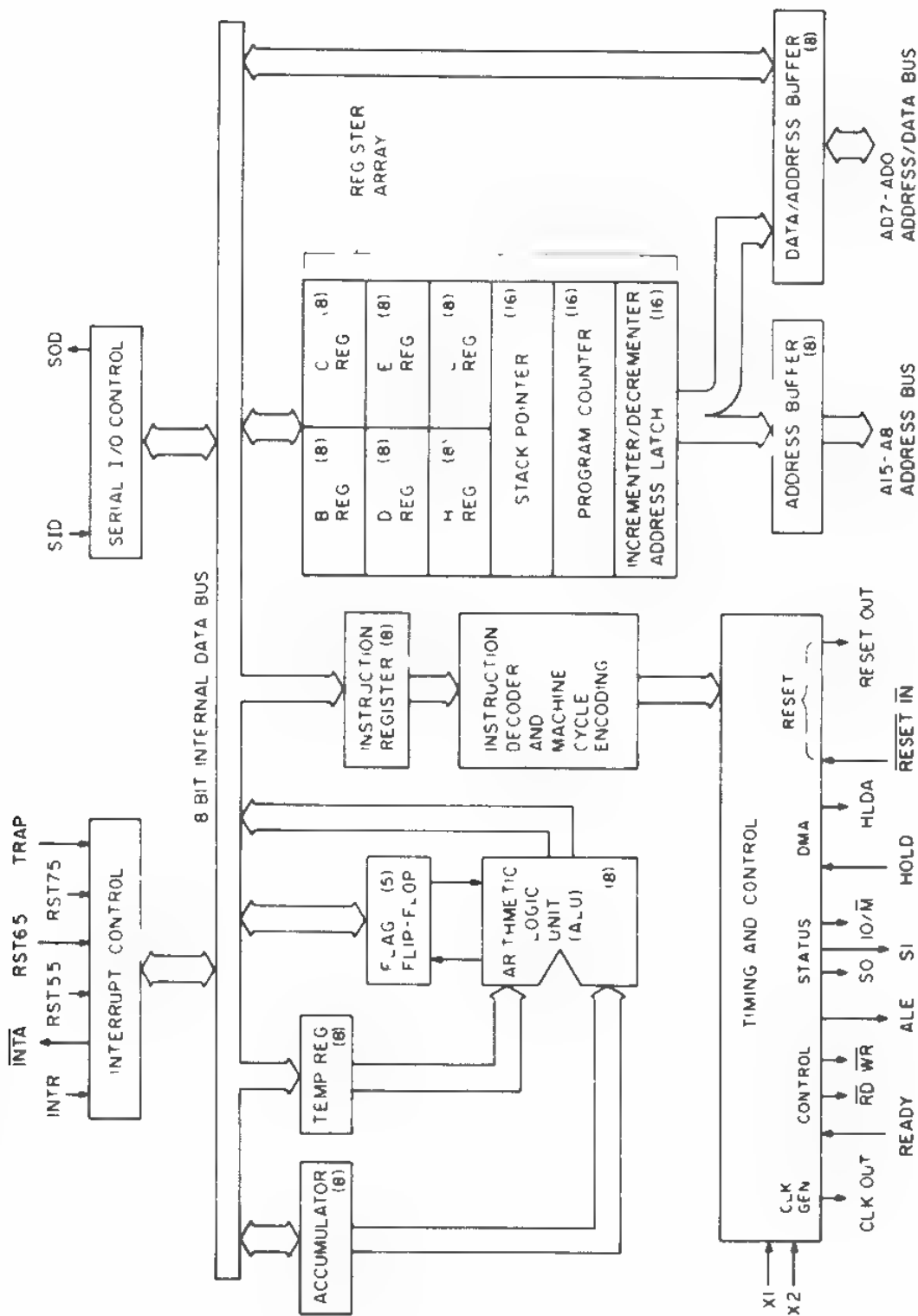


Figure 2.1. CPU Architecture

Within the CPU, various registers contain ones and zeros. These can be manipulated by the programmer through selected opcodes. The registers that are used most often are the A, B, C, D, E, F, H, L, SP, and PC registers.

The registers designated by a single letter are eight bits in size, while those designated by two letters are sixteen bits in size. Consistent with this convention, the eight-bit registers may be paired up, creating the AF, BC, DE, and HL registers, often called register pairs.

### **The Accumulator and the Flag Register**

The A register, also called the accumulator, is probably the most frequently used of the 8085 registers. Most of the other registers cannot be used for arithmetic calculations. Only the accumulator has an arithmetic logic unit (ALU) for addition, subtraction, multiplication, and logical AND and OR functions. Only the accumulator can be loaded to and from the input and output ports. Whenever an arithmetic operation is performed in the accumulator, one or more bits of the F or *flag* register will be affected. The Carry flag, for instance, is said to have been *set* if its new value is one or *reset* if its new value is zero.

Set and reset are also used to refer to individual bits elsewhere in the Model 100, including the I/O ports. "Turn on bit 4", "set bit 4," and "make bit 4 equal to one" are synonymous; so are the terms "turn off bit 0," "reset bit 0," and "make bit 0 equal to zero."

**Table 2.1.** F register

Bit	Symbol	Meaning
0	CY	Set if a <i>carry</i> resulted from bit 7 during an addition or a <i>borrow</i> resulted from bit 7 during a subtraction. Otherwise reset.
2	P	Set if the parity (number of ones in the accumulator) is even. Otherwise reset.
4	AC	Set if a <i>carry</i> resulted from bit 3 during an addition. Otherwise reset.
6	Z	Set if the value in the accumulator is zero. Otherwise reset.
7	S	This is a copy of bit 7 from the accumulator, often used to represent the "sign" of a small integer.

The flags of the F register are laid out as shown in table 2.1. These flags do not change every time the contents of the accumulator change. For example, if the Z flag is set, loading a nonzero value into the accumulator does not, in and of itself, reset it. The flags are set only during certain arithmetic and logic operations.

The condition of a particular flag is most often used as the determining factor in a conditional jump. The contents of the flag register can also be treated as an eight-bit byte and loaded to other registers through the PUSH and POP instructions.

## The Other Registers

The B, C, D, E, H, and L registers can be used as general-purpose storage locations. It is easy to move information among them as well as between them and the accumulator.

In addition, they can be used as register pairs. BC and DE are general-purpose register pairs, while HL can be used for a variety of addressing techniques, determining the address in memory to and from which information can be transferred.

When the HL register is used for this purpose, H contains the high-order portion of the address (bits 8 through 15), while L contains the low-order portion (bits 0 to 7).

The PC, or program counter, contains the numerical value of the address containing the opcode that is currently being executed. Usually the PC slowly increases by one address at a time, but its value changes drastically when, for example, a jump instruction is executed.

The SP, or stack pointer, is used whenever subroutine calls and returns occur. It manages an area of RAM called the *stack*. As will be seen in the discussion of the CALL, RET, PUSH and POP instructions, the stack is a LIFO (last-in, first-out) storage device. The design of the 8085 is such that items are placed on the *bottom* of the stack. As the stack grows, it grows into lower addresses of RAM.

## Clock Frequency and Execution Speed

From time to time, it is interesting to know just how long it takes for the 8085 to execute an opcode, a subroutine or an entire program. This is helpful in deciding which of several programming techniques will be faster, and is crucial in the design of certain routines which perform time-sensitive input or output functions.

The rate at which the Model 100 executes opcodes is determined by the crystal frequency provided to the 8085 CPU. In the Model 100 it is a 4.9152 Megahertz (cycles per second) crystal designated X2. It happens that the 8085 divides this frequency by two, yielding 2.4576 MHz, and uses that lower frequency to time the opcodes.

At 2.4576 cycles per second, each cycle is about 0.407 microseconds long. Each opcode requires a specified number of cycles to complete. These numbers are listed in this chapter. (For those who have worked with the 8080 or Z80, be careful, as the 8085 cycle times are sometimes different.)

## The Opcodes

The most often used opcodes are the *move* or *load* instructions, which do just what you might think they would do.

These instructions have the 8080 mnemonic MOV r,r' (for the word *move*) and Z80 mnemonic LD r,r' (for the word *load*). The 8080

and Z80 mnemonics are discussed in detail in chapter 4. In binary, the opcode is "01dddsss" where ddd is a three-bit representation of the destination register, and sss is a three-bit representation of the source register.

The three-bit values are:

111	A
000	B
001	C
010	D
011	E
100	H
101	L

The opcode 57, or 01010111, causes the contents of the A register (value 111) to be moved to the D register (value 010). The contents of the source register are unchanged. The corresponding mnemonics are MOV D,A and LD D,A.

Seven of the forty-nine possible moves are quite uninteresting, since they accomplish nothing -- such as a move from D to D or from A to A for example. The interesting moves are listed in table 2.2. The MOV instruction requires four clock cycles. This is different from the 8080 which requires five cycles.

Register moves have no effect on the flag register. No single instruction allows loading to or from the F register.

**Table 2.2.** 8085 register moves

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic
120	78	MOV A,B	LD A,B
121	79	MOV A,C	LD A,C
122	7A	MOV A,D	LD A,D
123	7B	MOV A,E	LD A,E
124	7C	MOV A,H	LD A,H
125	7D	MOV A,L	LD A,L
71	47	MOV B,A	LD B,A
65	41	MOV B,C	LD B,C
66	42	MOV B,D	LD B,D
67	43	MOV B,E	LD B,E
68	44	MOV B,H	LD B,H
69	45	MOV B,L	LD B,L
79	4F	MOV C,A	LD C,A
72	48	MOV C,B	LD C,B
74	4A	MOV C,D	LD C,D
75	4B	MOV C,E	LD C,E
76	4C	MOV C,H	LD C,H
77	4D	MOV C,L	LD C,L
87	57	MOV D,A	LD D,A
80	50	MOV D,B	LD D,B
81	51	MOV D,C	LD D,C
83	53	MOV D,E	LD D,E
84	54	MOV D,H	LD D,H
85	55	MOV D,L	LD D,L
95	5F	MOV E,A	LD E,A
88	58	MOV E,B	LD E,B
89	59	MOV E,C	LD E,C
90	5A	MOV E,D	LD E,D
92	5C	MOV E,H	LD E,H
93	5D	MOV E,L	LD E,L
103	67	MOV H,A	LD H,A
96	60	MOV H,B	LD H,B
97	61	MOV H,C	LD H,C
98	62	MOV H,D	LD H,D
99	63	MOV H,E	LD H,E
101	65	MOV H,L	LD H,L
111	6F	MOV L,A	LD L,A
104	68	MOV L,B	LD L,B
105	69	MOV L,C	LD L,C
106	6A	MOV L,D	LD L,D
107	6B	MOV L,E	LD L,E
108	6C	MOV L,H	LD L,H



## Memory Moves

One-byte moves of information can also be performed between a register and any location in memory. The location in memory is determined or *pointed to* by the HL register. As a result, the Z80 mnemonic for this instruction includes the symbol (HL), which means the location in memory whose address is in HL.

LD E, (HL) or MOV E,M, where M stands for memory, takes the contents of the memory location determined by HL and places it in E. The contents of the source code register or source memory location remain unchanged. These data transfers require seven clock cycles.

This is an example of so-called *indirect addressing*, in which the memory location involved is determined by the contents of a register pair.

The memory move opcodes take the form "01dddsss" where ddd or sss are 110 when referring to (HL). Otherwise the memory move opcodes obtain their value as in the Mov r,r' instruction above. The mnemonics are listed in table 2.3.

**Table 2.3.** Moves to and from memory

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic
126	7E	MOV A,M	LD A, (HL)
70	46	MOV B,M	LD B, (HL)
78	4E	MOV C,M	LD C, (HL)
86	56	MOV D,M	LD D, (HL)
94	5E	MOV E,M	LD E, (HL)
102	66	MOV H,M	LD H, (HL)
110	6E	MOV L,M	LD L, (HL)
119	77	MOV M,A	LD (HL),A
112	70	MOV M,B	LD (HL),B
113	71	MOV M,C	LD (HL),C
114	72	MOV M,D	LD (HL),D
115	73	MOV M,E	LD (HL),E
116	74	MOV M,H	LD (HL),H
117	75	MOV M,L	LD (HL),L

In addition to data transfers in which the source is a register or memory location, it is possible to load into a register or memory location from the opcode itself. The eight-bit value to be sent to the destination is simply the second byte of a two-byte opcode. The data

transfer is often called *immediate*, a term which is meant to convey that the byte comes directly (*immediately*) from program memory. The 8080 mnemonic "MVI" stands for *move immediate*. These instructions appear in table 2.4. In each case the value loaded may be any eight-bit value. Here and throughout the chapter the expression "FF" will be used to represent any eight-bit value, and "FFFF" will be used to represent any sixteen-bit value. Recall that when assembled into machine code, a value such as 7F34 becomes 347F in the opcode.

This instruction is analogous to a numerical constant in a BASIC program. Each instruction listed in the table requires seven clock cycles, except the load to (HL), which requires ten cycles.

**Table 2.4.** Eight-bit immediate load

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic
62	3E FF	MVI A,FF	LD A,FF
6	06 FF	MVI B,FF	LD B,FF
14	0E FF	MVI C,FF	LD C,FF
22	16 FF	MVI D,FF	LD D,FF
30	1E FF	MVI E,FF	LD E,FF
38	26 FF	MVI H,FF	LD H,FF
46	2E FF	MVI L,FF	LD L,FF
54	36 FF	MVI M,FF	LD (HL),FF

### Other Eight-bit Moves

All other eight-bit data transfers are limited to the accumulator as either source or destination. These transfers are listed in table 2.5.

The LDA and STA instructions are an example of so-called *direct addressing*, in which the memory location involved in the transfer is indicated by the second and third bytes of a three-byte opcode. The address comes *directly* from the program being executed. Each takes thirteen clock cycles.

The STAX and LDAX instructions allow data transfer to and from the memory location pointed to by the BC or DE register pair. This is register indirect addressing. Each instruction requires seven clock cycles.

The IN and OUT instructions cause an eight-bit word to be transferred between the accumulator and an input or output port.

These instructions are discussed at length in chapter 5. The IN and OUT instructions also constitute direct addressing, because the port address involved in the data transfer is determined directly by the program. Each instruction requires ten cycles.

**Table 2.5.** Other eight-bit data transfers

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic
58	3A FF FF	LDA FFFF	LD A,(FFFF)
10	0A	LDAX B	LD A,(BC)
26	1A	LDAX D	LD A,(DE)
50	32 FF FF	STA FFFF	LD (FFFF),A
2	02	STAX B	LD (BC),A
18	12	STAX D	LD (DE),A
219	DB FF	IN FF	IN A,(FF)
211	D3 FF	OUT FF	OUT (FF),A

## Sixteen-bit Data Transfers

The 8085 instruction set also allows sixteen-bit data transfers, as listed in table 2.6. The LHLD and SHLD instructions, for load HL direct and store HL direct, transfer two bytes of information to or from two adjacent locations in memory. The second and third bytes of the three-byte instruction determine the memory address to or from which L is loaded. The H register is loaded to or from the next higher memory address. The addressing is direct because the memory locations involved are specified directly by the program. Each instruction requires sixteen clock cycles.

The LXI (for load extended immediate) allow the loading of a sixteen-bit value from the program to a register pair. Ten clock cycles are required.

**Table 2.6.** Sixteen-bit data transfers

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic
42	2A FF FF	LHLD FFFF	LD HL,(FFFF)
34	22 FF FF	SHLD FFFF	LD (FFFF),HL
1	01 FF FF	LXI B,FFFF	LD BC,FFFF
17	11 FF FF	LXI D,FFFF	LD DE,FFFF
33	21 FF FF	LXI H,FFFF	LD HL,FFFF
49	31 FF FF	LXI SP,FFFF	LD SP,FFFF

One lone instruction is available for an exchange of registers: the XCHG or EX DE, HL, with decimal value 235 and hex value EB. It exchanges the contents of the HL and DE registers. This instruction, by far the briefest of the sixteen-bit transfers, requires only four clock cycles.

## Stack Operations

When you write a machine-language program, the most frequent reminders of the existence of the stack are the PUSH and POP instructions. The stack is also affected during subroutine CALLs and RETurns. Be sure there is always a POP for every PUSH, and vice versa.

When a PUSH is executed, the contents of the specified register pair are placed on the stack.

The contents of the high-order register of the pair (A, B, D, or H) are moved to the memory location one lower than the location pointed to by SP. The contents of the lower-order register are moved to the memory location two positions below the location pointed to by SP. The contents of SP are decremented by two. Twelve clock cycles are required to execute a PUSH.

The POP instruction undoes the action of the PUSH instruction. Ten clock cycles are required. The stack-related opcodes are listed in table 2.7.

The PUSH and POP instructions provide the only means of loading the entire flag register. For example, PUSH AF followed by POP BC moves the flag register contents into the C register and in the process, destroys whatever was previously stored in the B and C registers.

The 8080 mnemonic uses the rather cryptic abbreviation PSW (processor status word) to refer to the AF register pair.

The SPHL instruction moves the contents of HL to the SP register. Six clock cycles are required.

The XTHL instruction exchanges the contents of the HL register with the location pointed to by SP. More specifically, the contents of L are exchanged with the contents of the memory location pointed to by SP, and the contents of H are exchanged with the contents of the memory location one address higher than that pointed to by SP. Sixteen clock cycles are required.

**Table 2.7.** Stack-related opcodes

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic
193	C1	POP B	POP BC
209	D1	POP D	POP DE
225	E1	POP H	POP HL
241	F1	POP PSW	POP AF
197	C5	PUSH B	PUSH BC
213	D5	PUSH D	PUSH DE
229	E5	PUSH H	PUSH HL
245	F5	PUSH PSW	PUSH AF
249	F9	SPHL	LD SP,HL
227	E3	XTHL	EX (SP),HL

## Branch Instructions

A variety of instructions are available to transfer control; each causes the PC to do something other than simply increment.

The JMP (jump) instruction loads a new value into the PC. The conditional jump instructions test the condition of one or more flags and load a new value into the PC only if the condition is satisfied. In each case, the new PC value is contained in the second and third bytes of a three-byte opcode. This is shown by the value FFFF in table 2.8. Each jump requires ten cycles, unless a conditional jump fails to satisfy its condition, in which case, seven cycles are required.

The flag requirements for the various conditions are given in table 2.8.

**Table 2.8.** Jump instructions

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic
195	C3 FF FF	JMP FFFF	JP FFFF
218	DA FF FF	JC FFFF	JP C,FFFF
210	D2 FF FF	JNC FFFF	JP NC,FFFF
250	FA FF FF	JM FFFF	JP M,FFFF
242	F2 FF FF	JP FFFF	JP P,FFFF
234	EA FF FF	JPE FFFF	JP PE,FFFF
226	E2 FF FF	JPO FFFF	JP PO,FFFF
202	CA FF FF	JZ FFFF	JP Z,FFFF
194	C2 FF FF	JNZ FFFF	JP NZ,FFFF

**Table 2.9.** Conditional execution

Condition	Flag Contents to Satisfy the Condition
C-carry	CY=1
NC-no carry	CY=0
P-positive	S=0
M-minus	S=1
PE-parity even	P=0
PO-parity odd	P=1
Z-zero	Z=1
NZ-nonzero	Z=0

## Subroutine Calls

The call and return instructions share with the jump instructions the ability to change the PC and thus to redirect the flow of program execution. Just as in a BASIC subroutine call, the 8085 CALL instruction transfers control to a specified address. The place where the call occurred is noted, so that when the subroutine finishes (returns), control can be returned there.

When a CALL opcode is encountered, the PC is incremented so that it points to the next executable instruction following the opcode that was the CALL. The PC value is placed on the stack just as if it were PUSHed there. The second and third bytes of the CALL instruction are treated as an address and placed in the PC. Execution continues based on the PC contents.

Later, when a RET instruction is encountered, the stack is “popped” and the sixteen-bit value on the stack is placed in the PC. Execution continues based on the PC contents. The call and return require eighteen and ten cycles, respectively.

The 8085 subroutine instructions use the stack as defined by the SP, just as the POP and PUSH instructions do. As a result, programming errors can occur causing the return instruction to load a meaningless value into the PC. The 8085 may start executing code that is not even a program. At best the program will not execute properly; at worst everything in RAM may be lost, and you will find yourself back at January 1, 1900.

Two precautions will keep you out of trouble. Be sure the number of PUSHes and POPs encountered in the execution of the subroutine are always equal, regardless of any internal branching, and never tamper with the SP register.

## Conditional Calls and Returns

The 8085 recognizes conditional subroutine calls and returns. The conditions that can be tested for are the same as the conditions listed in table 2.9. If the condition fails, execution proceeds to the next instruction, much the same as in the case of a conditional jump.

A conditional call requires eighteen clock cycles if the condition is satisfied, and nine cycles otherwise. A conditional return requires twelve cycles if the condition is satisfied, and six cycles otherwise.

The subroutine opcodes are listed in table 2.10.

**Table 2.10.** Subroutine opcodes

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic
205	CD FF FF	CALL FFFF	CALL FFFF
220	DC FF FF	CC FFFF	CALL C,FFFF
212	D4 FF FF	CNC FFFF	CALL NC,FFFF
252	FC FF FF	CM FFFF	CALL M,FFFF
244	F4 FF FF	CP FFFF	CALL P,FFFF
236	EC FF FF	CPE FFFF	CALL PE,FFFF

*continued on following page*

228	E4 FF FF	CPO FFFF	CALL PO,FFFF
204	CC FF FF	CZ FFFF	CALL Z,FFFF
196	C4 FF FF	CNZ FFFF	CALL NZ,FFFF
201	C9	RET	RET
216	D8	RC	RET C
208	D0	RNC	RET NC
248	F8	RM	RET M
240	F0	RP	RET P
232	E8	RPE	RET PE
224	E0	RPO	RET PO
200	C8	RZ	RET Z
192	C0	RNZ	RET NZ

### Restart Instructions

The 8085 also responds to eight “call” opcodes each of which is a single byte long, one-third the length of the usual subroutine call. These instructions are listed in table 2.11, and require twelve clock cycles each to execute.

**Table 2.11.** Restart instructions

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic	Location of Called Subroutine
199	C7	RST 0	RST 00	0000
207	CF	RST 1	RST 08	0008
215	D7	RST 2	RST 10	0010
223	DF	RST 3	RST 18	0018
231	E7	RST 4	RST 20	0020
239	EF	RST 5	RST 28	0028
247	F7	RST 6	RST 30	0030
255	FF	RST 7	RST 38	0038

Restart instructions allow the eight most frequently used subroutines in ROM to be assigned to these one-byte opcodes. This saves program space every time the routine is called. In a machine other than a Model 100, the restart instructions allow an interrupting device to “jam” a single byte onto the data bus in such a way that the single byte



can determine which of several routines will be used to handle the interrupt. (The latter process is described further in chapter 15.)

These one-byte subroutine calls, called RST (for restart) instructions, are dedicated to eight predetermined addresses near 0000H. Because that part of address space is in ROM and thus cannot be changed by the user, the user cannot change what happens when one of the RST instructions is executed.

Nevertheless, the ROM code associated with the RST instructions may be put to use; the ROM functions are listed in table 2.12.

**Table 2.12.** Model 100 restart routines

Mnemonic	Jumps to	Function
RST 00	7D33	Same as RESET
RST 08		Find any BASIC special character
RST 10	0858	Find next BASIC character
RST 18		Compares DE and HL
RST 20	4B44	Sends character to LCD or PRT
RST 28	1069	Determines variable type
RST 30	33DC	Returns sign of first floating point accumulator
RST 38	7FD6	Indexed jump (see text)

The RST 20 instruction is handy. It sends a character to the screen or printer depending on the condition of an output flag in RAM (see chapters 10 and 13).

The RST 18 routine compares the DE and HL registers. Upon return from the routine, the Z flag is set if the registers are identical, and reset otherwise. To see how this is done, disassemble the code at 0018 to 001D.

The RST 00 instruction accomplishes the same thing as a jump to 0000, although in fewer bytes. RST 00 causes an initialization of the kind that occurs when the RESET button is pressed or the power to the Model 100 is turned on.

The RST 38 instruction is best thought of as a two-byte instruction. The byte following the RST is used as an offset. It points to a two-byte address in a table located in RAM at addresses FADA through FB39, and that address is jumped to. This may be compared with the ON ... GOTO command in BASIC.

The RST 38 and RAM table are designed to allow for expansion. Many BASIC commands contain an RST 38, and in each case, the address in the RAM table returns (in cases where an existing Model 100 feature may change) or generates an FC error (in cases where a feature is to be added). Particular RST 38 applications are described in later chapters.

If you want to test your understanding of the stack to the fullest, disassemble and study the code at 7FD6 through 7FF3, which accomplishes the vectored jump of the RST 38 subroutine.

The RST 08 instruction can also be thought of as a two-byte instruction. The byte following the RST 08 opcode is treated as an ASCII character or BASIC token. It is compared with the next character in a BASIC program line. One odd occurrence is the use of the exchange instruction at 0009. The exchange is performed between HL and the memory location pointed to by SP, but usually that memory location is in ROM. The instruction is essentially used as a one-way data transfer, since you cannot successfully load data into ROM.

The RST 10 routine is used in parsing a BASIC program line. It locates the next significant character, ignoring spaces, tabs, and the like.

The RST 28 and 30 routines are used in handling variables. RST 28 determines the variable type, and RST 30 returns the sign of the value in the first floating-point accumulator.

### **Another Jump Instruction**

The 8085 CPU recognizes a jump instruction that might be thought of as a sixteen-bit register load, since it loads the contents of the HL register into the PC. It is analogous to the BASIC ON ... GOTO command. The decimal value for the opcode is 233 (hex E9). The 8080 mnemonic is PCHL, and the Z80 mnemonic is JP (HL).

## Arithmetic Functions

The 8085, like all microprocessors, has rather limited arithmetic capabilities. It can add or subtract, if the numbers are not too large, and it can multiply and divide, if the multiplier or divisor is a power of two. Anything else must be performed as a combination of the above instructions.

The arithmetic and logic functions of the 8085 set and reset the various flags in the F register. The first and most straightforward of the arithmetic functions is addition. Several kinds are provided for. In each case, one of the two numbers to be added is placed in the accumulator. After the instruction has been executed, the result is found in the accumulator.

The other of the two numbers to be added can be found in another register (register direct addressing), somewhere in memory (register indirect addressing), or in the latter part of a two-byte instruction (immediate). The available opcodes are listed in table 2.13.

**Table 2.13.** Eight-bit Add instructions

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic
135	87	ADD A	ADD A,A
128	80	ADD B	ADD A,B
129	81	ADD C	ADD A,C
130	82	ADD D	ADD A,D
131	83	ADD E	ADD A,E
132	84	ADD H	ADD A,H
133	85	ADD L	ADD A,L
134	86	ADD M	ADD A,(HL)
198	C6 FF	ADI FF	ADD A,FF

The ADD (and its relatives, the subtract, add-with-carry, and subtract-with-borrow) direct instructions require four cycles, while the indirect and immediate instructions require seven cycles. In each case, all of the flags in the F register are updated, based on the final condition of the accumulator.

The Carry (C or CY) and Auxiliary Carry (AC) flags deserve special explanation.

When two numbers are added, the sum is often too large to fit into the accumulator. When this happens, the carry (CY) flag is set, and a properly written program checks to see if a carry occurred and responds accordingly.

Similarly, when dealing with BCD (binary-coded decimal, explained in chapter 11) numbers, you may want to know whether the result is too large to fit in four bits. If a carry occurred from bit 3 to bit 4, the auxiliary carry flag (also sometimes known as the half-carry), is set. It is reset otherwise.

The AC flag cannot be used as the condition of a jump or call. It is used by the DAA instruction.

### Addition with Carry

When numbers that do not fit into a single accumulator are being manipulated, you must break them into parts and treat them separately. When two such numbers are being summed, you add from right to left, just as you were taught in grade school.

When the rightmost part is added, a carry may occur. You need a way to add that carry to the part of the number that has not yet been summed (the part to the left of the part that has been summed). The 8085 instruction set specifically provides for this with the “add-with-carry” opcodes, which are listed in table 2.14. The add-with-carry instructions are identical to the ADD instructions in all respects except for the handling of the carry flag.

**Table 2.14.** Eight-bit Add-with-carry

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic
143	8F	ADC A	ADC A,A
136	88	ADC B	ADC A,B
137	89	ADC C	ADC A,C
138	8A	ADC D	ADC A,D
139	8B	ADC E	ADC A,E
140	8C	ADC H	ADC A,H
141	8D	ADC L	ADC A,L
142	8E	ADC M	ADC A,(HL)
206	CE FF	ACI FF	ADC A,FF

## Subtraction

The subtract and subtract-with-borrow instructions work the same way as the add and add-with-carry instructions. The two carry flags have analogous meanings. In the case of subtraction, the CY flag indicates whether a *borrow* was attempted from bit 8. This happens when a larger number is subtracted from a smaller one. The AC flag indicates whether a borrow was made from bit 4. The subtract and subtract-with-borrow instructions appear in table 2.15. In the case of the subtract-with-borrow, the borrow flag is subtracted from the accumulator as part of the subtraction process.

**Table 2.15.** Subtraction instructions

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic
151	97	SUB A	SUB A,A
144	90	SUB B	SUB A,B
145	91	SUB C	SUB A,C
146	92	SUB D	SUB A,D
147	93	SUB E	SUB A,E
148	94	SUB H	SUB A,H
149	95	SUB L	SUB A,L
150	96	SUB M	SUB (HL)
214	D6 FF	SUI FF	SUB FF
159	9F	SBB A	SBC A,A
152	98	SBB B	SBC A,B
153	99	SBB C	SBC A,C
154	9A	SBB D	SBC A,D
155	9B	SBB E	SBC A,E
156	9C	SBB H	SBC A,H
157	9D	SBB L	SBC A,L
158	9E	SBB M	SBC A,(HL)
222	DE FF	SBI FF	SBC FF

In table 2.15, the 8080 mnemonics SUI and SBI stand for *subtract immediate* and *subtract-with-borrow immediate*, while the Z80 mnemonic SBC stands for *subtract-with-carry*.

## Eight-bit increments and decrements

The 8085 can be instructed to increment (increase by one) or decrement (decrease by one) an eight-bit register or the contents of a memory address. All condition flags except CY are affected. In the case of the registers, the instruction requires four cycles, while the memory increment or decrement requires ten cycles. These instructions are listed in table 2.16.

**Table 2.16.** Eight-bit increments and decrements

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic
60	3C	INR A	INC A
4	04	INR B	INC B
12	0C	INR C	INC C
20	14	INR D	INC D
28	1C	INR E	INC E
36	24	INR H	INC H
44	2C	INR L	INC L
52	34	INR M	INC (HL)
61	3D	DCR A	DEC A
5	05	DCR B	DEC B
13	0D	DCR C	DEC C
21	15	DCR D	DEC D
29	1D	DCR E	DEC E
37	25	DCR H	DEC H
45	2D	DCR L	DEC L
53	35	DCR M	DEC (HL)

## Binary-Coded Decimal Operations

One arithmetic instruction is used solely for binary-coded decimal operations. After addition has taken place, it can be used to clean up the accumulator so that neither the upper half nor the lower half of the accumulator contains the BCD equivalent of a number greater than nine.

This is what happens when the DAA (decimal adjust accumulator) instruction is executed. If the low-order four bits constitute ten or larger or if the AC flag was set previously, the value six is added to the accumulator. This may cause a carry into bit 4.

If the high-order four bits have a value of ten or more or if the CY flag was set previously, the value six is added to the high-order four bits of the accumulator. This may cause a carry into bit 8 to occur.

This opcode has a value of 39 decimal, or 27 hex, and requires four clock cycles. All condition flags are affected. The opcode is used in six different places in the ROM BASIC arithmetic routines, between 2C34 and 2E40.

## Sixteen-bit Arithmetic

The sixteen-bit capability of the 8085 is quite limited. Addition can be performed, and the HL register is used as the accumulator. Prior to the summation, one of the addends is located in the HL register, and the other is located in another register pair. When the addition is finished, the sum is completed in the HL register.

Of the various condition flags, only the CY flag is affected by sixteen-bit arithmetic. It is set if the sum yields a carry at bit 15. The instruction requires ten clock cycles. The instructions are listed in table 2.17. The mnemonic DAD stands for *double add*.

**Table 2.17.** Sixteen-bit addition

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic
9	09	DAD B	ADD HL,BC
25	19	DAD D	ADD HL,DE
41	29	DAD HL	ADD HL,HL
57	39	DAD SP	ADD HL,SP

Sixteen-bit increment and decrement instructions are also available. These are listed in table 2.18. Each requires six clock cycles; no condition flags are affected. The mnemonics INX and DCX stand for *increment extended* and *decrement extended* respectively.

**Table 2.18.** Sixteen-bit increments and decrements

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic
3	03	INX B	INC BC
19	13	INX D	INC DE
35	23	INX H	INC HL
51	33	INX SP	INC SP
11	0B	DCX B	DEC BC
27	1B	DCX D	DEC DE
43	2B	DCX H	DEC HL
59	3B	DCX SP	DEC SP

## Logical Operators

The 8085 is designed to perform logical operations on eight-bit values. This is useful not only in calculations but also as a way of setting a particular bit equal to 1 or 0.

The AND, OR, and XOR instructions are listed in table 2.19. In each case, one number is placed in the accumulator, the operation is performed, and the result is found in the accumulator. The operations take place exactly as described on page 111 of the Model 100 Owner's Manual. The other number can be found in another register. It can be a value found at a memory address, or it can be a constant stored as the second byte of a two-byte instruction (immediate).

The immediate and memory operations require seven clock cycles, and the register operations require four cycles.

**Table 2.19.** AND, OR, and XOR operations

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic
167	A7	ANA A	AND A
160	A0	ANA B	AND B
161	A1	ANA C	AND C
162	A2	ANA D	AND D
163	A3	ANA E	AND E
164	A4	ANA H	AND H

*continued on following page*



Decimal	Hex	8080 Mnemonic	Z80 Mnemonic
165	A5	ANA L	AND L
166	A6	ANA M	AND (HL)
230	E6 FF	ANI FF	AND FF
183	B7	ORA A	OR A
176	B0	ORA B	OR B
177	B1	ORA C	OR C
178	B2	ORA D	OR D
179	B3	ORA E	OR E
180	B4	ORA H	OR H
181	B5	ORA L	OR L
182	B6	ORA M	OR (HL)
246	F6 FF	ORI FF	OR FF
175	AF	XRA A	XOR A
168	A8	XRA B	XOR B
169	A9	XRA C	XOR C
170	AA	XRA D	XOR D
171	AB	XRA E	XOR E
172	AC	XRA H	XOR H
173	AD	XRA L	XOR L
174	AE	XRA M	XOR (HL)
238	EE FF	XRI FF	XOR FF

With the AND, OR, and XOR instructions, the carry (CY) flag is cleared. The OR and XOR operations clear the AC flag as well.

The manner in which the AND instruction handles the AC flag differs between the 8080 and the 8085. This is one of the few substantive differences between the two CPUs that might make an 8080 program run incorrectly on an 8085. In the case of the 8085, the AND operation simply turns the AC flag on. The 8080, however, sets it equal to bit 3 of the result of the AND operation in the accumulator.

### **TURNING A BIT ON**

For example, to turn on bit 2 of the accumulator, use the OR instruction with a value of two to the power of two, namely 4.

### TURNING A BIT OFF

To turn off bit 3 of the accumulator, use the AND instruction with a value that has all bits on except bit 3, namely 11110111, or FC hex.

### TESTING A REGISTER PAIR FOR ZERO

Often you want to know whether the value in a register pair, such as BC, has been decremented to zero. The easiest way to do this is to load B into the accumulator, and OR it with the C register.

### COMPARISON OPERATIONS

Sometimes you want to know whether two values are equal, but the algebraic difference of the two is not required. The compare instruction determines the relation between two values. It sets the condition flags at the values they would have if the two numbers had been subtracted. The value in the accumulator is unchanged as a result of the operation.

In particular, the Z flag is set at 1 if the two compared values are equal. The carry (CY) flag is set at 1 if the accumulator contents are less than the other value. The latter result is obtained because if the operation had been a subtraction and if a larger value were subtracted from the accumulator, a borrow would have occurred. The compare instructions are listed in table 2.20.

**Table 2.20.** Compare instructions

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic
191	BF	CMP A	CP A
184	B8	CMP B	CP B
185	B9	CMP C	CP C
186	BA	CMP D	CP D
187	BB	CMP E	CP E
188	BC	CMP H	CP H
189	BD	CMP L	CP L
190	BE	CMP M	CP (HL)
254	FE FF	CPI FF	CP FF

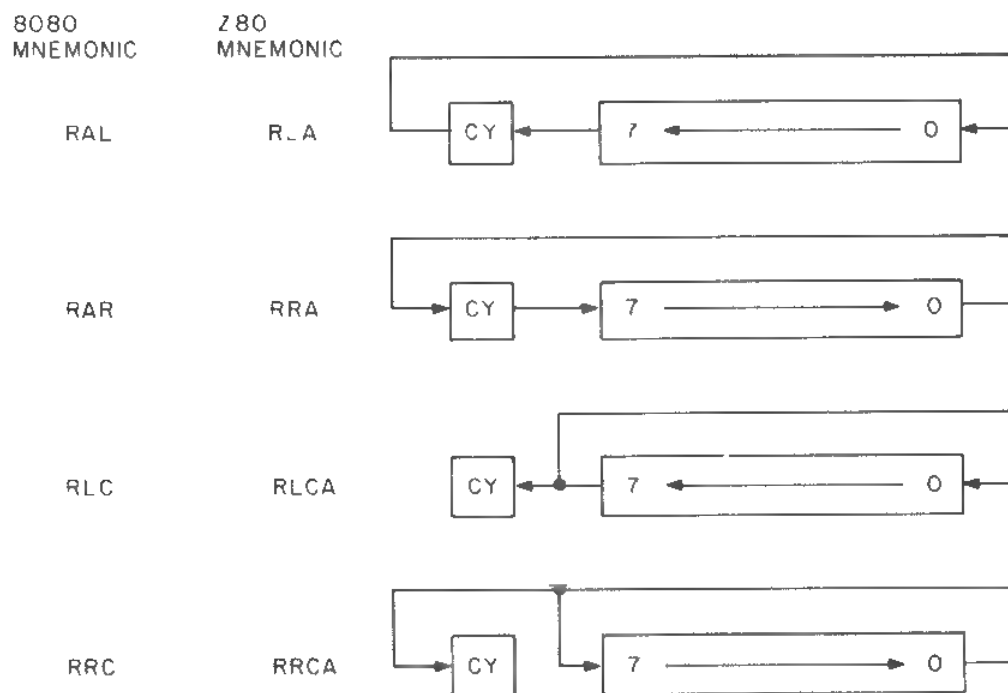
## Rotate Instructions

The rotate instructions are easier to illustrate than to describe (*see* figure 2.2.) A rotate instruction shifts the entire contents of the accumulator one position to the left or right, leaving an open bit at the empty end of the accumulator. In the RLCA and RRCA instructions, what goes out one end of the accumulator comes back in the other end and goes to the carry. In the RLA and RRA instructions, what goes out one end of the accumulator ends up in the carry, while the contents of the carry come in the empty end of the accumulator. An RLA followed by an RRA leaves everything as before, with no loss of information. In contrast, either of the RLCA and RRCA instructions destroy the former contents of the carry flag. The opcodes and mnemonics are listed in table 2.21.

**Table 2.21.** Rotate instructions

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic
23	17	RAL	RLA
31	1F	RAR	RRA
7	07	RLC	RLCA
15	0F	RRC	RRCA

Each instruction requires four clock cycles, and in each case only the carry flag is affected.



**Figure 2.2.** Rotate instructions

## USES FOR THE ROTATE INSTRUCTIONS

Rotation to the left or right is comparable to multiplying or dividing by two, respectively.

If a jump that is conditional on either bit 7 or 0 of the accumulator is desired, the most economical way to program it is to rotate the bit of interest into the carry, and then jump conditional on the carry flag.

When serial information is being sent to the CPU, whether as an input on the SID pin (*see* chapter 12), or through an I/O port (*see* chapter 11), the rotate instructions provide a handy way to reconstruct the byte:

- rotate the recently received bit into the carry
- load the partial byte (from the previous bits received) into the accumulator
- rotate the carry bit into the accumulator
- repeat the process until eight bytes have been loaded.

A similar process may be used to send out serial data from the CPU, as discussed in chapters 11 and 12.

## Other Logical Instructions

The *accumulator complement* instruction turns off each bit of the accumulator that is on, and turns on each bit that is off. This is also known as a *ones complement*. No condition flags are affected. The operation requires four clock cycles.

The opcode and mnemonics are shown in table 2.22.

**Table 2.22.** Other logical operations

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic	Meaning
55	37	STC	SCF	set carry flag
63	3F	CMC	CCF	complement carry flag
47	2F	CMA	CPL	complement accumulator

Because of the way negative numbers are treated in addition and subtraction, simply complementing the accumulator is not the way to change the sign on an integer stored there. Instead a *two's complement* must be performed. A two's complement consists of the performance of a one's complement followed by the addition of 1.

## The Carry Flag

The carry flag may be turned on by means of the *set carry flag* instruction or may be complemented by the *complement carry flag* instruction. The only condition flag affected is the carry flag. Each operation requires four clock cycles.

## Machine Control Instructions.

The HALT instruction stops the processor. This is used in ROM as part of the power-down sequence at 1431-1458, and also appears enigmatically in the TEXT program at 6AC3.

The only way to overcome a halt is by a RESET or by turning the power off and on again.

NOP stands for *no operation* and is just that -- an instruction which does nothing except use up four clock cycles. No flags are affected. Sometimes a NOP is placed in RAM to allow new instructions to be inserted later without the need to reassemble the program.

**Table 2.23.** Machine control opcodes

Decimal	Hex	8080 Mnemonic	Z80 Mnemonic
243	F3	DI	DI
251	FB	EI	EI
118	76	HLT	HALT
0	00	NOP	NOP
32	20	RIM	(none)
48	30	SIM	(none)

The EI and DI instructions are used to enable and disable interrupts, respectively. This is discussed in detail in chapter 15. Interrupts are disabled when a time-sensitive process is to be performed or when a particular sequence of loading registers or output ports must not be interfered with.

The RIM (read interrupt mask) and SIM (set interrupt mask) instructions are used to perform cassette input and output (*see* chapter 12) and to mask selected interrupts (*see* chapter 15).

# 3

---

## Advanced BASIC

---

There are a number of BASIC commands and functions which, though only briefly explained in the owner's manual, must be fully understood before one can begin machine language programming with the Model 100. These are covered in detail in this chapter.

Recall from the discussion in chapter 1 that the Model 100 architecture allows the 8085 CPU to communicate with the integrated circuits around it by means of I/O ports (of which 18 are implemented out of a possible 256) and memory locations, all of which are implemented.

In assembly language the I/O ports are accessed by means of IN and OUT opcodes, while memory locations are accessed by means of load, store, and move opcodes. In BASIC the I/O ports are accessed by means of the INP function and OUT command, while the memory



locations are accessed by means of the PEEK function and POKE command. This is summarized in table 3.1.

**Table 3.1.** Machine language and BASIC data transfer

Access to	Direction	BASIC Method	Typical Machine Language	
			8080	Z80
Input port	To CPU	INP(n)	IN n	IN A,(n)
Output port	From CPU	OUT n,d	OUT n	OUT (n),A
ROM, RAM	to CPU	PEEK(n)	MOV r,M LDA n LDAX	LD A,(n)
RAM	from CPU	POKE n,d	MOV M,r STA n STAX n	LD (n),A

In addition to the differences in terminology between assembly language and BASIC operations, there is a syntactical difference. In assembly language an opcode is an opcode. In BASIC, however, some reserved words (like PEEK and INP) denote functions, forming part or all of an expression, and thus never follow a line number or colon; while others (like POKE and OUT) are functions and always follow a line number or a colon.

## PEEK

The PEEK function is executed with one argument, the memory address, which is an integer in the range 0 to 65535. The value returned by the function is the contents of that memory location, an integer in the range 0 to 255. PEEKs are always harmless -- executing a PEEK will not alter the memory contents.

The PEEK function returns an eight-bit integer, but often one wishes to use PEEKs to obtain a sixteen-bit integer (such as the HIMEM value, discussed below) from two adjacent memory addresses. Because the 8085 stores such integers with the high-order eight bits in the higher memory address, the proper way to combine the PEEK values is to multiply the higher PEEK by two to the eighth power, or 256.

For example, to return the sixteen-bit value at 64430, evaluate,

`PEEK(64430)+256*PEEK(64431)`

Many locations worth peeking to (or at) are listed in figure 18.4.

## **POKE**

The POKE command has two operands. The first, an integer in the range 0 to 65535, specifies the memory address to which data is written, and the second, an integer in the range 0 to 255, is the data value to be written.

Although the BASIC interpreter allows POKES to locations below 32768, such pokes have no effect, because those locations are in ROM. (If the standard ROM were switched out, as described in chapter 5, and if RAM were put in its place, then a POKE to that part of memory would accomplish something.)

Using POKES blindly can be dangerous. Many pointers, flags, and the like, stored in the area 62960-65535 are essential to the operating system, and if changed indiscriminantly, can cause destruction of user files. This destruction may not occur until several days later, when a particular file or routine is accessed.

Some values in the 62960-65535 area, however, such as the SOUND flag, may be freely changed by means of a POKE command. These *safe* addresses are discussed in later chapters in the context of particular Model 100 functions.

Even if POKES are confined to RAM below 62960, problems can still arise. A BA file, for instance, contains two-byte addresses for succeeding BASIC line numbers. If a line number address is changed, the BASIC interpreter's reaction when that program is run (which might be several days later) is unpredictable.

DO files are somewhat safer. As long as one stays away from the beginning and end-of-file bytes, one may change values freely. The worst that can happen is that the file, when later viewed through TEXT or OPENed in BASIC, will be found to have the wrong contents.

CO files contain load addresses which, if tampered with, may cause a later LOADM or RUNM to yield unpredictable addresses.

Often one wishes to store a sixteen-bit value in RAM. This is accomplished in a manner similar to a sixteen-bit PEEK. The low-order eight bits are stored to the named location, and the high-order eight bits are stored to the next higher memory location.

For example, to store the value 62700 to memory address 62964, one must separate the low-order and high-order parts of the integer 62700. In principle this could be accomplished as follows:

high-order part is  $62700 \text{ AND } (255 * 256)$   
low-order part is  $62700 \text{ AND } 255$

Unfortunately, BASIC will not perform AND, OR and XOR operations on integers larger than 32767. One way to avoid this problem is to test the desired value, say 62700, and if it is bigger than 32767, simply subtract 65536 from it, yielding in this case -2836. Because BASIC does operate on integers between -32767 and 0, they may be used.

Thus the high-order part is  $(-2836 \text{ AND } (255 * 256))$ , and the low-order part is  $(-2836 \text{ AND } 255)$ . The proper POKEs are:

POKE 62694,  $(62700 - 65536) / 256$   
POKE 62965,  $(62700 - 65536) \text{ AND } 255$

Long division can also be used to separate the high and low-order parts. For an integer N, the high-order part H is  $H = \text{INT}(N / 256)$ , and the low-order part L is the remainder, namely  $L = N - (H * 256)$ .

## Input Ports

The various active input ports are surveyed in chapter 5, and discussed in detail in later chapters relating to particular devices. The INP function has one argument, the port number, which is an integer in the range 0 to 255. The value returned by the function is the data from that input port.

When an input is attempted from a port that has not been implemented in hardware, the value returned turns out to be the number of the port. This happens because of the shared address and data lines in the 8085.

Use of the INP function, like the PEEK function, is always harmless — it is impossible to hurt anything.

## OUT

The OUT command, like the POKE command, has two arguments. It uses the second argument (an integer between 0 and 255) as the outgoing data. The first argument, also an integer in the range 0 to 255, is the port number to which the data is to be sent. The various output port numbers implemented in the Model 100 are surveyed in chapter 5 and discussed in detail in later chapters.

OUT commands can be dangerous, though not, perhaps, as dangerous as POKES. Sending the wrong value to an output port can select the option ROM when you don't want it (any port in the range 224-239) or can cut off power in a disorderly way (port 178 or 186). Output to other port numbers is generally harmless, although this may, for example, upset the UART settings.

## ALLOCATING RAM — THE MAXRAM AND HIMEM VALUES

As shown in figure 1.2, RAM is partitioned into areas with distinct uses. The lowest RAM address (32768 decimal or 8000 hex in a 32K RAM model) is stored in RAM at 64192. Recall from our earlier discussion that this value may be determined by evaluating:

```
PEEK(64192)+256*PEEK(64193)
```

BA files are placed starting at the lowest RAM address. The top address for the last BA file is stored at 64430. This value, and all the other "top addresses" discussed below, are updated every time a BA program changes in size, is created through the SAVE command, or killed. DO files are placed above BA files in RAM. The top of the DO files is pointed to by 64432. That pointer, and other pointers described below, are updated when a DO file grows, shrinks, or is killed.

CO files are next with the top of the CO files pointed to by 64434. That pointer is updated when a CO file is created or destroyed.

The BASIC variables, string and numeric, are stored above the CO files. These are set whenever a BASIC program is being run.

Each block of data is "bumped upward" when more data is stored below. For instance, if a BASIC program writes data to a DO file,

several items move up in memory: the top boundary of the DO files, the entirety of the CO files, and the BASIC variable area.

As more and more data is stored, sooner or later, one is greeted with the "OM" error, which means out of memory. This is because the types of data described so far consume all of the space between 8000 hex (in a 32K machine) and HIMEM, the user-imposed limit on data storage.

## **HIMEM**

HIMEM is a special BASIC variable which may form part of an expression (e.g. PRINT HIMEM-4) but cannot appear to the left of an equal sign. HIMEM is stored at 62964, and so may be changed by POKEing to 62964 and 62965. (See the preceding discussion of POKEing of sixteen-bit values).

The more conventional way to change HIMEM is using BASIC's CLEAR command.

## **CLEAR**

The CLEAR command, when executed in BASIC, always clears string and numerical variables and closes all files. Any dimensioned variables lose their dimensioned status.

By convention the CLEAR command should appear at the beginning of a program. For example, run this program:

```
10 F=25:DIM G(100)
20 CLEAR
30 PRINT F:G(99)=45:PRINT G(99)
```

The value for F will be displayed as 0, not 25, as F was CLEARED in line 20. The attempt to set G(99) equal 45 will be met with a "BS" (bad subscript) error, as the CLEAR command's execution in line 30 caused G to lose its dimensioned status.

Because of the way BASIC handles variables, it needs to know ahead of time how much space will be needed for string variables. Usually BASIC reserves 256 bytes, but this value may be changed by providing an argument for the CLEAR command. (Values smaller

than 256 can be specified. This can be helpful if additional numeric variable space is required.)

The unused, reserved string variable space can be determined by evaluating the function FRE(""). String constants do not occupy the string variable space as shown in the following program:

```
10 CLEAR:PRINT FRE (""):I$="GHJKGKJHG":  
PRINT FRE (""):I$="GHJKGKJHG"+"":PRINT FRE ("")
```

The first value printed will be 256, the number of bytes set aside for string variables. When the variable I\$ is assigned a value, one might think doing so would consume some of the variable space. BASIC, however, simply stores a pointer to the place in the BA program (much lower in RAM) where the string constant assigned to I\$ can be found. The free space is still 256.

When I\$="GHJKGKJHG"+" is executed, BASIC is forced to evaluate a string expression, and I\$ must point to the result. The only place to store the result is in the string variable space. As a result the free space is diminished.

By executing the FRE function with a numeric argument, e.g. FRE(0) or FRE(4.4), the amount of unused BASIC numeric space can be determined.

The CLEAR command may also have a second argument, namely a user-selected value for HIMEM. As mentioned above, HIMEM sets an upper limit on how high in memory user data storage (BA, DO, and CO files, and BASIC variables) may expand. As a result, anything above the address stored in HIMEM will generally be undisturbed.

The largest permissible value stored in HIMEM is F5F0, the bottom address of the operating system RAM area. This value, F5F0, is available to BASIC in the special variable MAXRAM. MAXRAM, like HIMEM, cannot appear on the left side of an equals sign. If and when a disk operating system is installed on the Model 100, the value returned by MAXRAM will be smaller, probably E000 hex (57344 decimal).

The second argument of the CLEAR function is checked before BASIC changes HIMEM. If an integer larger than MAXRAM is

given, an "FC" (function call) error will result, while an attempt to set it lower than the top of BASIC variables will yield an "OM" (out of memory) error.

To remove any protection given earlier to high memory, simply type `CLEAR 256,MAXRAM`. This does not immediately destroy data in the protected area; it simply renders it vulnerable in the event that BA, DO, or CO files, or BASIC variables expand upward to fill that area.

### **SAVEM,CSAVEM**

Four BASIC commands, `CLOADM`, `CSAVEM`, `LOADM`, and `SAVEM` have been provided to facilitate manipulation of machine-language programs.

Upon generating machine language instructions, the assembler places these in RAM, usually at the location where it is intended that the machine language program will run. In the Model 100, this is usually a location in RAM between `HIMEM` and `MAXRAM`. The block of memory containing the program has a starting and an ending address, and the program itself usually has a entry (or transfer) address.

(Although assemblers vary, usually the start address is set by an `ORG` or `ORIGIN` psuedo-opcode, and the entry address is set by an `END` psuedo-opcode.)

The `SAVEM` command may be used to store the program as a RAM file, freeing the protected RAM area for other uses. `SAVEM`'s syntax is:

`SAVEM strexp, stadd, endadd, tradd`

The argument `strexp` is a string expression containing a filename preceded optionally by a device. The device may be `CAS:` or `RAM:`. If none is specified, RAM is assumed. The filename will have the extension `".CO"` appended. The arguments `stadd` and `endadd` are integers specifying the starting point and ending point in RAM of the block to be stored in the CO file.

The CO file that results from the command contains not only the data from high RAM, but also information about where the data will be reloaded if the file is accessed using menu selection (discussed later) or LOADM. As discussed in chapter 18, the CO file contains the address to start reloading to (always the same as the start address when the SAVEM was performed); the size of the program (obtained by subtracting endadd from stadd); and the transfer address. This applies to both tape and RAM files.

Note that after a SAVEM to RAM, two copies of the machine language program now exist — one in high memory and one somewhere between the DO files and the BASIC variable space. If memory space is short, one may free up the high RAM by typing CLEAR 1,MAXRAM.

The CSAVEM command functions like the SAVEM command except that a device of CAS: is automatically prefixed to the filename.

### **LOADM AND CLOADM**

Assuming a tape or RAM file with extension CO has been created, the machine language program may be reloaded to protected RAM by using the BASIC command LOADM. The syntax of the command is:

LOADM strexp

where strexp is a string expression containing a filename and optionally a device type of CAS: or RAM:. (If no device is specified, RAM: is assumed.)

First, the CO file, which may originate from tape or RAM, is opened. The proposed start address for reloading is compared with HIMEM. The file is loaded only if its start address is greater than HIMEM. The purpose of this check is to avoid damage to user files in the area between 8000 hex and HIMEM.

Before executing LOADM, then, it may be necessary to use the CLEAR command to reset HIMEM to a low enough value to accommodate the CO file.

If LOADM is executed rather than the program mode, the start, end, and entry addresses will be displayed on the screen.



It should be clear from this discussion that the BASIC commands SAVEM and LOADM only allow the program to be loaded to its original location.

The BASIC command CLOADM works exactly like LOADM except that a device type of CAS: is prefixed to the filename.

### **VARPTR**

One unpublished aspect of the VARPTR function is that if a file number is used as the argument, the value returned is the address of the file control block. Further discussion would be beyond the scope of this book.

### **CALL**

BASIC's CALL performs essentially like the machine language CALL. If and when the called routine returns, control is returned to the calling BASIC program.

The syntax of the CALL function is as follows,

CALL add,exp1,exp2

where add is required, but exp1 and exp2 are optional.

The argument add is an integer between 0 and 65535 and is the address called. Thus the called routine may be located in ROM or RAM. One would expect that the only RAM addresses called would be those in the area HIMEM and MAXRAM, since there is usually no other executable code in RAM. (There is machine code in CO files but it cannot be CALLED from BASIC as the jump addresses within the CO file do not correspond to the place in RAM where the CO file resides.)

The argument exp1 is an integer between 0 and 255, and is placed in the accumulator just prior to the call. The argument exp2 is an integer between -32768 and 65536, and is converted to a sixteen-bit integer and placed in the HL register-pair. (See the previous discussion of BASIC's treatment of sixteen-bit integers.)

Often one wishes to use HL as a pointer to a series of memory locations containing ASCII characters, for printing or other processing. This can be accomplished by placing the characters into a string variable and using the `VARPTR` function to determine `exp2`. For example if the string is `F$`, its location is the sixteen-bit integer at `VARPTR(F$)+1`. In other words, if `B=VARPTR(F$)`, then `exp2` should be `PEEK(B+1)+256*peek(B+2)`.

Calls can be dangerous. The called routine is surely not as fully error-protected as BASIC itself, and data stored in RAM may be disturbed. Make frequent backups of user files until you are certain the called routine is safe as these may be destroyed.

The `CALL` command provides one of two easy ways of executing user-written machine language programs. Assuming the machine language program is located somewhere between `HIMEM` and `MAXRAM`, one may run a short BASIC program to `CALL` the machine language program. Often a program contains both a BASIC portion and a machine language portion. For instance the BASIC portion could be used to prompt to open a file, and the machine language portion could be used to search RAM or ROM for a certain value or read a cassette tape in non-standard format. The BASIC program first sets `HIMEM`; then it uses `LOADM` to place the machine program into high memory. Then, with one or more `CALLs` to one or more addresses in the machine language program, the "dirty work" is done.

### **MENU SELECTION OF CO FILES**

The Model 100 operating system is designed to respond to menu selection not only of BA files (by running them) and DO files (by entering `TEXT`), but also of CO files. When the cursor is moved to a CO file name and `ENTER` is pushed, the CO file is loaded to high memory, assuming that the current value of `HIMEM` leaves enough room for the file. (If `HIMEM` is too high in value, the operating system will beep and return to the menu.) Any machine language program previously located in the protected high RAM area will be destroyed.

Assuming an entry or transfer address was specified when the CO file was created, CPU execution will proceed at that address.

For this reason menu selection of a CO file can be dangerous. Unless you are quite certain that the CO program behaves itself and does not tamper with the wrong parts of RAM memory, be prepared for loss of user files.

### **BATTLE OF THE CO FILES**

Z80 programmers are accustomed to so-called relative jumps, which make it possible to write programs which will run no matter where in RAM they have been loaded. Unfortunately, the 8085 has no relative jump instructions in its instruction set. Each jump must contain the actual sixteen-bit address to which control will be transferred. Unless part of the machine language program actually resides at that location, the program will not function properly.

No problem arises when a BASIC program uses a single CO routine and no other, since the BASIC program may LOADM the CO file into high memory and use it, paying no attention to the previous contents.

### **POKING INTO PROTECTED MEMORY**

BASIC itself can be used both to load and run a machine language program. Obviously this is practical only if the program to be run is small, because no one enjoys typing in long BASIC DATA statements.

But in situations where it is intended that a routine, perhaps a bar-code-reader device handler, will remain in high RAM indefinitely, there will be a conflict. The cleanest, but perhaps most troublesome, solution is to reassemble the other machine routine so that it may reside in RAM below the bar-code-reader driver, and protect both routines.

Any two companies offering a line of machine language software for the Model 100 will likely choose loading addresses designed to conflict with each other's loading addresses, so that whichever company makes the first sale to a customer will be likely to make all future sales. The customer may need to disassemble and reassemble such routines to change their load locations.

Program development is tedious since even the smallest program change necessitates recalculating and retyping numerous decimal integers.

Nonetheless, without access to an assembler, BASIC POKEs are really the only choice. Also, sometimes machine language is intended for just one little piece of a program while BASIC is used to receive the user input or to format the output. Such a BASIC program must first use the CLEAR command to protect the portion of high memory where the machine language code will be placed. Then a FOR loop is used to load the values into memory. If desired, a CALL command may then be used to execute the machine language.

The following example demonstrates how to use BASIC POKEs to accomplish machine language programming. Note how similar the ADDRESS and SCHEDUL programs are — they scan the ADRS and NOTE files, respectively, and display or print records containing a particular word or phase. The entry addresses for the two programs may be found in the system file directory (described in chapter 18). These are 5B68 and 5B6F.

If you disassemble the code at 5B68-5B72, you will see that each entry point leads to 5B74, the “plain vanilla” program underlying the two. Prior to reaching 5B74, each entry point sets DE to point to an ASCII string of the name of the file to be scanned and sets the accumulator with a flag value to indicate whether the command prompt will be “Adrs:” or “Sched:”.

Your first reaction might be to simply use the BASIC CALL command to call 5B74. Unfortunately the CALL command cannot be used to set the DE register, only the HL. Furthermore, to keep the stack in order, a jump to 5B74 must be executed rather than a call.

Thus the way to harness the ADDRESS/ SCHEDL program for your own use is to set HL to point to the filename, and then execute the following machine code:

D1		POP DE
EB		EX DE, HL
C3	74 5B	JMP 5B74

The purpose of the first POP is to eliminate the return address from the CALL, because it will never be used. EX represents a simple method of loading the value in HL into the DE register. A jump to the ROM program follows.

After writing the assembly language program, you must assemble it. This can be done by hand using the opcode tables in the appendices. The decimal values appear in the CHR\$ functions.

The rest of the program retrieves the user input, converts it to uppercase, and provides the pointer to the location of the opcodes for the CALL. The program in finished form follows:

```
20 P$=CHR$(209)+CHR$(235)+CHR$(195)+CHR$(116)+CHR$(91):PRINT"Input file:
";:LINEINPUTF$:F$=LEFT$(F$,6)+".DO"+CHR$(0):FORI=1TOLEN(F$):C=ASC(MID$(F$,I,1)):IFC>96 AND C<123THENC=C-32
120 POKE64984+I,C:NEXT:A=VARPTR(P$)
CALLPEEK (A+1) +256*PEEK(A+2),0,64985
```

# 4

---

## Borrowing From Z80 Experience

---

### **Differences Between The Z80 and 80C85 Instruction Sets**

Recall from the discussion in chapter 2 that the most machine level programming is accomplished by writing a program in human-readable *assembly mnemonics* which are processed by an assembler to yield hexadecimal, and ultimately binary, values which are stored in RAM or ROM and then executed by the CPU.

Of the 256 possible values which may be returned when the CPU fetches an eight-bit byte from memory, 244 have precisely the same result when executed by 8080, 8085 and Z80 processors. Two hundred of them comprise one-byte opcodes, eighteen serve as the first byte of two-byte opcodes, and twenty-six serve as the first byte of three-byte opcodes. These values are listed in appendix C, tables C.1, C.2, and C.3.

Twelve eight-bit values (08, 10, 18, 20, 28, 30, 38, CB, D9, DD, ED, and FD), however, are treated differently by the three processors. The behavior of the 8080 is undefined for all twelve; they may be thought of as *gaps* in the 8080 instruction set. (There are even more gaps, so defined, in the 8008 instruction set.)

The designers of the 8085 filled in two of the twelve gaps: 20H and 30H are one-byte instructions which load the interrupt mask register (present in the 8085 but not in the 8080 or Z80) to and from the accumulator (A) register. The behavior of the 8085 is undefined as to the remaining ten values; this is shown by the data entries in tables C.1, C.2, and C.3. Nonetheless, any program written for the 8080 will run on the 8085, since the permissible instructions for the 8085 merely expand upon, but do not change, the 8080 instruction set.

The designers of the Z80 filled in all twelve gaps in the 8080 instruction set, but filled them in differently than did the designers of the 8085. Thus, while any program written for the 8080 will run on the Z80, not every program written for the 8085 will run on the Z80. More to the point, for Model 100 owners as a general rule programs written for the Z80 will not run on the 8085.

Because of the close relationship between the Z80 and 8085 instruction sets, programming techniques and algorithms from Z80 machines are nonetheless helpful in writing and modifying programs for the Model 100. Indeed, many readers of this book first learned assembly language programming with the Z80 microprocessor, since it was used in the Radio Shack TRS-80 Model I, III, and IV computers. Thus it is instructive to devote some attention to the Z80 instruction set.

Let's compare, for example, the behavior of the 8085 and Z80 CPUs upon fetching the value 7EH in the course of executing a program. By referring to table C.1, you can see that a programmer accustomed to the 8085 would think of this as a MOV A,M instruction, while a Z80 programmer will consider it to be a LD A,(HL) instruction. The result in each case is the same — the contents of one of the 65536 memory locations available to the CPU (and pointed to by the 16-bit register HL) are moved (loaded) to the A register. The only difference is a semantic one — the assembly mnemonic read by an assembler is MOV A,M according to the conventions for mnemonics

set up by Intel when it introduced the 8008 and 8080 processors, or LDA, (HL) according to the conventions set up by Zilog when it introduced the Z80.

Likewise an analogy may be drawn between most of the Z80 and 8085 mnemonics. Table C.2 lists the 8085 mnemonics alphabetically with the corresponding Z80 mnemonic. Similarly, table C.3 lists the Z80 mnemonics alphabetically with the corresponding 8085 mnemonic.

As should be apparent from this discussion, those Z80 operations whose opcodes appear in the gaps of the 8080 instruction set do not appear in the 8085 instruction set. They would result in undefined behavior if executed by the 8085.

When converting a Z80 program for 8085 execution, it is not just a matter of finding a combination of 8085 instructions to replace each non-8085 Z80 instruction. This is because the Z80 contains several registers (IX, IY, AF', BC', DE', HL', refresh register R, and interrupt page register I) not present in the 8080 or 8085. The Z80 instructions manipulating these registers have no close substitute in the 8080 or 8085 instruction sets.

Many other non-8085 instructions, though, have relatively easy substitutes in the 8085. For example, in the 8085:

- There are no bit set or test instructions, though of course the equivalent can always be accomplished through the AND, OR, and rotate instructions.
- There are no relative jumps JR nor DJNZ. Many Z80 programs are location-independent through use of relative jumps, while any jump instruction for the 8085 must provide the absolute address.
- The various incremented and decremented loads and compares are missing. These include: LDI, LDIR, LDD, CPI, CPID, CPD, and CPDR. These routines may all be performed piecemeal by combinations of simpler 8085 instructions.
- The NEG instruction, which performs a 2's complement on the A register and sets various flags, is not available. The former may be performed by taking the 1's complement CPL (CMA) and adding one, but it is important to realize the CPL instruction only sets the H and N flags.



- Two Z80 16-bit arithmetic instructions are missing: ADD HL,ss and SBC HL,ss. The ADD HL,ss (DAD) instruction is a close but not perfect substitute because it sets only the H,N and C flags.
- Rotate instructions (RLC, RL, RRC, RR) may be applied only to the A register. Thus if it is desired to rotate the contents of some other register, it will be necessary to load the register to the accumulator, rotate it, and load the contents back to the other register.
- The shift instructions of the Z80: SLA, SRA, and SRL, are not available. The accumulator rotate instructions must be used instead, masking the left and rightmost bits as necessary with AND or OR instructions.
- The Z80 double-word rotate instructions (RLD and RRD) are not available. Single-word rotate instructions must be used instead.
- The following Z80 I/O instructions are not available in the 8085: IN r,(C), OUT (C),r, INI, INIR, IND, INDR, OUTI, OTIR, OUTD, and OTDR. In the 8085, input and output ports may be loaded to and from the A register only, and the port number must appear directly as part of the two-byte opcode.
- The interrupt modes of the 8085 are determined through the SIM instruction, rather than the Z80 opcodes IM 0, IM 1, and IM 2. Return from an interrupt is accomplished with a simple RET rather than RETI or RETN.

The following general principles, then, should guide you in converting a Z80 program for Model 100 use:

1. I/O locations in the Model 100 are all different than for any other machines. (See table 5.4.) In particular, the TRS-80 Models I, III, and IV accomplish some I/O through memory-mapped devices accessed by LD instructions rather than IN and OUT instructions.
2. Any Z80 instruction starting with 08, 10, 18, 20, 28, 30, Z8, CB, D9, DD, ED, or FD must be replaced somehow.
3. Subroutine calls to ROM will be different in the Model 100 than in any other machine.
4. References to IX, IY, AF', BC', DE', HL', refresh register R, and interrupt page register I must be replaced somehow.
5. Relative jumps (JR) must be converted to absolute jumps (JP).

If you have a Z80 assembler operating on another machine (such as a Model I, III, and IV) you can use it as an aid in Model 100 assembly programming, at least as a convenient way to assemble mnemonics into hex code. (Strictly speaking, this is called using the assembler as a *cross-assembler*.) The following points should be considered:

1. Always print and examine a listing of the assembly process to be sure you have not used *forbidden* opcodes starting with Z8, 10, 18, 20, 28, 30, 38, CB, D9, DD, ED, and FD. Your listing should show no four-byte opcodes, as there are no four-byte opcodes in the 8085 instruction set.
2. If you are using RIM or SIM instructions, you will have to insert them manually, holding their place in the mnemonic source with NOP instructions.
3. If the assembler accepts 8080 mnemonics, you may wish to learn and assemble them instead of Z80 mnemonics, as this provides a built-in safeguard against using *forbidden* opcodes.

With a bit of ingenuity you may be able to work out a way to load the hex values from the assembly machine into the Model 100.



# 5

---

## Understanding the Hardware of the Model 100

---

The Model 100 uses an 80C85 microprocessor. The letter “C” indicates that it is a CMOS version of the 8085 device, which means that it draws very little power. Most of the integrated circuits within the Model 100 are CMOS devices. This was done to conserve battery power. In this and subsequent chapters, 8085 is used to reference this microprocessor rather than 80C85, since the two devices are so similar.

Communication between the 8085 CPU and the rest of the world is accomplished almost exclusively through the input/output ports. The 8085 has four ways of communicating with the circuitry:

- the 65536 memory addresses which are used in the Model 100 for RAM and ROM access
- the 256 I/O ports, of which twelve are currently used in the Model 100
- the serial input and output pins, used in the Model 100 for cassette I/O
- the interrupt pins which are used for various purposes.

## Memory Locations

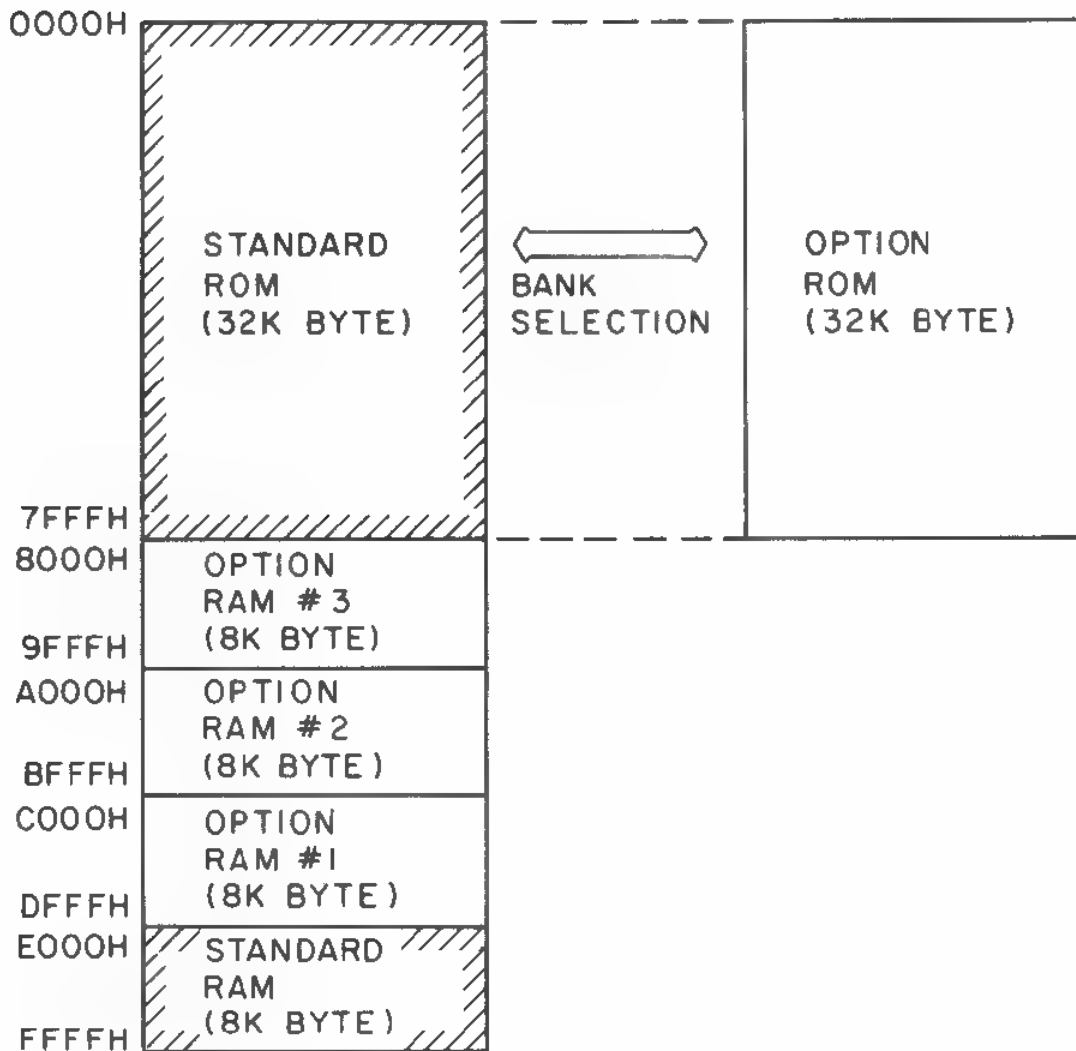
The CPU has the ability to load to and from a large number of memory addresses, selected by turning on and off combinations of the sixteen address lines. The number of distinct addresses is two to the sixteenth power, or 65536.

In the Model 100, the bottom half of this so-called address space contains read-only memory or ROM. Depending on certain port outputs, memory accesses within this part of the address space connect with the standard ROM chip M12 or a chip in an optional ROM socket M11.

In other words, PEEKs to addresses below 32768 yield one set of values if the standard ROM is selected, and another set of values if the option ROM is selected. When the computer is turned on, it sets itself to the standard ROM. (The option ROM socket is discussed in chapter 17.)

Read-only memory, as its name suggests, cannot be written to. If you try to change its contents, by means of a POKE in BASIC or a store instruction in machine language, you will find its contents unchanged. Fortunately you cannot cause any harm to the ROM by doing this.

The top half of address space (numerically speaking) is set aside for RAM chips, shown in figure 5.1. An 8K machine (catalog no. 26-3801) has RAM soldered in place from E000 to FFFF with three sockets in the area from 8000 to DFFF. A 24K machine (catalog no. 26-3802) has RAM soldered in place from A000 to FFFF and a single socket for 8000 to 9FFF.



**Figure 5.1.** Memory map

Optional RAM modules can be installed. They can be plugged into any vacant sockets and the CPU will be able to access them, but the ROM operating system will only "discover" and use the RAM that extends in an unbroken series down from FFFF.

The serial input and serial output pins of the CPU, SID and SOD (pins 5 and 4 respectively) are used for cassette input and output. This is discussed in detail in chapter 12. The interrupt pins of the CPU (pins 6, 7, 8, 9, and 10) provide a variety of inputs to the CPU. They are discussed in detail in chapter 15. The cassette and interrupt pin assignments are shown in table 5.1.

**Table 5.1.** Input/ Output signals originating or terminating at the CPU chip

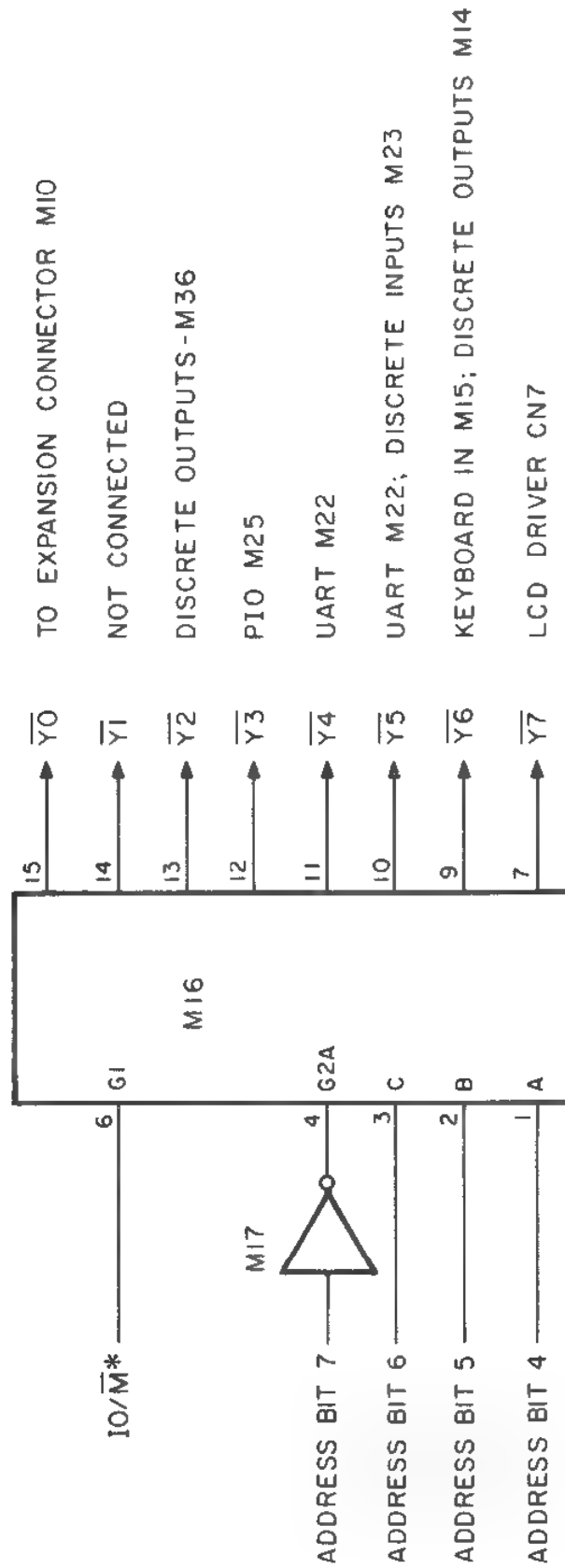
Pin	CPU Name	Signal	Function
4	SOD	SOD	Serial output to cassette
5	SID	SID	Serial input from cassette
7	RST 7.5	TP	256-Hertz pulse
8	RST 6.5	DR	UART data received
9	RST 5.5	BCR	Bar-code reader input
10	INTR	INTR	Expansion bus pin 17

When the 8085 executes an IN or OUT instruction, several things happen. The CPU asserts the IO/M\* line, indicating that it wants to talk to an I/O device rather than a memory chip. The eight-bit port address is made available on the address lines. Actually, the CPU offers the port address twice, once on the top half of the sixteen address lines, and again on the bottom half. Next, the CPU waits for an I/O device to offer or accept an eight-bit byte of data.

The eight address lines used to select I/O ports are connected to an *address decoder*, one or more integrated circuits whose task is to interpret the various possible combinations of off and on among the eight lines. Other devices are then activated accordingly. Most of the port address decoding duties in the Model 100 are performed by M16, which takes bits 7, 6, 5, and 4 of the port address as input and determines which port integrated circuit, if any, is activated. This is shown in figure 5.2.

The connection of the IO/M\* signal to M16 insures that M16 responds only to I/O port addresses and not to memory (RAM or ROM) addresses. The connection of address bit 7 causes M16 to respond only when bit 7 is on. In other words it responds only to port addresses above 127 (80H to FFH).

The connection of bits 4, 5, and 6 causes one of the eight outputs to be selected (pulled low) in response to the bit pattern. The chip-select signal Y0 is activated for any port address in the range of 80 to 8F. Y1 is activated for any address in the range of 90 to 9F, and so on up to Y7, which corresponds to F0 to FF.



**Figure 5.2.** Address decoding bits 7, 6, 5, and 4



The integrated circuits to which the Y signals are attached are listed in table 5.2, which shows the input devices, and table 5.3, which shows the output devices. (The "contents" column in table 5.3 is explained later.)

**Table 5.2.** Input port hardware

Y	Port	Integrated Circuit
Y3	BB	M25- PIO (port C)
Y4	C8	M22- UART (receiver buffer)
Y5	D8	M23- buffer
Y6	E8	M15- buffer
Y7	FE,FF	CN7- LCD connector

**Table 5.3.** Output hardware and access to contents

Y	Port	Integrated Circuit	Contents
Y2	A8	M36- flipflop	FAAE
Y3	B8	M25- PIO	Not available
Y3	B9	M25- PIO (port A)	Input port B9
Y3	BA	M25- PIO (port B)	Input port BA
Y3	BC,BD	M25- PIO (divider)	Not available
Y4	C8	M22- UART transmitter	Not available
Y5	D8	M22- UART control	Not available
Y6	E8	M14- Flipflop	FF45
Y7	FE,FF	CN7- LCD connector	

What about port address lines 0, 1, 2, and 3? M16 pays no attention to them. It has no way of knowing whether the CPU has requested a port input from 80, 81, 82, or any other value up to 8F. It activates Y0 for any of these addresses. This makes for a certain arbitrariness when you are writing a program. If the task is to get a byte of data from the UART, the result is the same whether you input from C0 or CF, or any value in between. This is because the UART itself ignores address bits 0 to 3. Throughout the ROM, the value C8 is used.

A few of the devices selected by M16 do pay attention to bits 0 to 3. For example, the PIO chip, selected by Y3, looks at bits 0 to 2 and responds differently, depending on which bit is on. B8, B9, BA, BB, BC, and BD are distinct port addresses in the Model 100. Taking into

account the various ways port address lines are connected in the Model 100, you can develop a map of “port space” somewhat like the address space, as shown in figure 5.1. This is depicted as a diagram in figure 5.2 and table 5.4.

**Table 5.4.** Port numbers

Port Number	Input Function	Output Function
00-6F	Not used	Not used
70-8F	See text	See text
90-9F	Hobby use	Hobby use
A8*	Not used	Phone/modem
B8 (&B0)	Not used	PIO divider control C3-start 43-stop
B9 (&B1)	Port contents	Parallel outputs: LCD, LPT, KB control, clock/calendar
BA (&B2)	Port contents	Output pins (see table 5.6)
BB (&B3)	Input pins (see table 5.7)	Not used
BC (&B4)	Not used	PIO divider lower byte
BD (&B5)	Not used	PIO divider upper byte and mode
B6,B7,BE,BF	Not used	Not used
C8*	UART in- coming data	UART outgoing data
D8*	input pins	UART control
E8*	Keyboard input	Output pins (see table 5.8)
FE*	LCD	LCD
FF*	LCD	LCD

\* Model 100 port addresses are not fully decoded. For example, all ports A0 through AF respond identically to A8.

## The Ports

No circuitry has been supplied to handle ports 00 to 6F, and nothing in the ROM suggests expansion in that area.

Ports 70 through 8F, though only partially implemented through the Y0 signal, appear to be intended for loading parallel data to and from some mass storage device plugged into the expansion bus connector M10. Perhaps these are part of the optional I/O control unit or RAM file unit referred to on page 4-12 of the service manual.

Ports 90 through 9F, which correspond to port-select signal Y1, are not connected to anything. This is described in the service manual as an optional telephone answering unit, and is discussed further in chapter 17.

Port A8 controls discrete functions. Bit 0 disconnects the telephone instrument and bit 1 enables modem carrier transmission. Usually you want to change only one of the bits. You can find the present contents of the port in RAM at FAAE, change bits using AND and OR operations, and write out to the port and to RAM.

Output port B8 programs the 81C55 PIO. The 81C55 PIO (Programmable Input/Output) chip is a forty-pin integrated circuit that does much of the I/O work of the Model 100. As it comes from the factory, it contains 256 bytes of RAM that never get used in the Model 100. It also contains three ports (A, B, and C) that are capable of being programmed as input or output ports, but the wiring of the Model 100 is such that ports A and B are always used for output (and their interrupt capability is never used), and C is always used for input. See table 5.5, which shows how the PIO discrete input and outputs are wired. Table 5.5 also shows other PIO connections.

Of the eight bits that can be output to port B8, six never change, as they would make the PIO do things the Model 100 wiring does not let it do. If you do inadvertently send the wrong values for these bits, no harm is done to the hardware.

Only two of the bits ever vary, bits 6 and 7. They control a so-called timer, which as used in the Model 100, would be better termed a divider. The divider-control bits 6 and 7 should be 11 (binary) to start the divider, and 10 to stop it. (This is explained in detail in chapters 7 and 9.)

**Table 5.5.** Signals originating or terminating at PIO

Pin	PIO Name	Signal	Function
1	PC3	BCR	Bar-code reader input
2	PC4	CTS	Clear-to-send
5	PC5	DSR	Data-Set ready
6	TO	RRC	UART receiver clock
6	TO	TRC	UART transmitter clock
8	CE	Y3	Ports 176-191 select
32	PB3	RS232C	RS232/modem select
33	PB4	PCS	Power control signal
35	PB6	DTRR	Data Terminal Ready
36	PB7	RTS	Request-to-send/off hook
37	PC0	DATAOUT	Clock/Calendar serial out
38	PC1	BUSYNOT	Line printer selected
39	PC2	BUSY	Line printer busy

Port B9 is the general-purpose parallel output, accomplished through PIO port A. It is used for the printer (*see* chapter 10), LCD (chapter 13), and keyboard (chapter 6). In addition, it is used to send serial data to the clock/calendar chip (chapter 11). Current contents of the port are obtained by reading from the port.

Port BA, like A8, controls discrete functions. Unlike A8, it is accomplished through the PIO chip (port B), so that current contents of the port are obtained by reading from the port. Bit 0 scans the keyboard modifier keys such as the shift and control keys. Bits 0 and 1 address the LCD. Bits 2 and 5 control the beeper (*see* chapter 9). Bit 3 switches from RS232 to modem mode (*see* chapters 7 and 8). Bit 4 removes power to the computer (*see* chapter 16). Bits 6 and 7 assert DTR and RTS when in RS-232 mode (*see* chapter 7). Bit 7 hangs up the phone when in modem mode (*see* chapter 8). This is shown in table 5.6.

Port BB is PIO input port C, used for sensing discrete signals. Bit 0 is clock/calendar data. Bits 1 and 2 are printer status. Bit 3 is bar-code reader input (chapter 14). Bit 4 is CTS or ANS/ORIG (chapters 7 and 8), and bit 5 is DSR or DIR/ACP (chapters 7 and 8). These input signals are shown in table 5.7.

**Table 5.6.** Output signals (output port BA)

Bit	Function
0	LCD control; keyboard scan e.g. SHIFT, NUM, CAPS
1	LCD control
2	Disconnect beeper from PIO divider
3	Switches from RS232 to modem
4	Power-control signal
5	Direct beeper control
6	DTR (0 yields + at RS-232 pin 20)
7	In RS-232 mode: RTS (0 yields + at RS-232 pin 4) in modem mode: phone line on-hook

**Table 5.7.** Input signals (input port BB)

Bit	Function
0	Clock/calendar data to CPU
1	LPT not busy (PRINTER pin 25)
2	LPT busy (PRINTER pin 21)
3	BCR input (1=ground at pin 2)
4	In RS232 mode (CTS; + at RS-232 pin 5 yields logic 0) in modem mode (1=ANS, 0=ORIG)
5	In RS232 mode (DSR; + at RS-232 pin 6 yields logic 0) in modem mode (1=ACP, 0=DIR)
6,7	Not used (always 1)

Ports BC and BD load the low and high bytes, respectively, of the divider used by the PIO to produce the baud rate (chapter 7) and beep frequency (chapter 9).

The UART sends and receives data through port C8 (chapter 7). Port D8, like port BB, provides discrete inputs to the CPU. Bit 0 is the carrier-detect signal (chapter 8). Bits 1, 2, and 3 indicate UART overrun, framing, and parity errors (chapter 7). Bit 4 is the UART transmitter buffer register empty signal (chapter 7). Bit 5 is the phone jack RP signal (chapter 17). Bit 6 is not fully implemented in hardware (see chapter 17), and bit 7 is the low-power signal (chapter 16).

The UART parameters (parity, word length, and so on) are programmed through output port D8 (chapter 7).

Output port E8 controls a number of discrete functions. Bit 0 selects the option ROM. Bit 1 strobes the printer. Bit 2 strobes the clock/calendar chip and bit 3 controls the cassette motor. You can find the present contents of the port in RAM at FAAE. These signals are shown in table 5.8.

**Table 5.8.** Output signals (output port E8; contents at FF45)

Bit	Function
0	STROM (1=select option ROM M11)
1	STROBE (1=ground at PRINTER pin 1)
2	STB (1=clock/calendar strobe)
3	REMOTE (1=CASSETTE pins 1 and 3 shorted)
4-7	Not used

Input port E8 provides a parallel input—the results of a keyboard scan. Input and output ports FE and FF are used for the liquid crystal display.

## Connectors in the Model 100

The connectors in the Model 100 are sometimes referred to by number, and are listed in order in table 5.9. In addition, the LCD printed circuit board contains connectors numbered CN1, CN2, and CN3. The acoustically-coupled modem contains two connectors each numbered CN1, and two connectors numbered CN2.

**Table 5.9.** Model 100 connectors

Connector	Function
CN1	Keyboard connector (chapter 6)
CN2	Bar-code reader connector (chapter 14)
CN3	Cassette connector (chapter 12)
CN4	Phone connector (chapter 8)
CN5	Printer connector (chapter 10)
CN6	RS-232C connector (chapter 7)
CN7	Liquid-Crystal Display connector (chapter 13) (corresponds to CN1 on LCD board)
CN8	Low Battery LED connector (chapter 16) (corresponds to CN3 on LCD board)
CN9	DC 6V connector (chapter 16)
M10	Expansion bus connector (chapter 17)
M11	Option ROM socket (chapter 17)

The Model 100 contains three relays, listed in table 5.10.

**Table 5.10.** Relays in the Model 100

Relay	Function
RY1	Cassette motor control (chapter 12)
RY2	Telephone off-hook (chapter 8)
RY3	Phone instrument relay (chapter 8)

The Model 100 contains three quartz crystals used for various time-sensitive functions; these are listed in table 5.11.

**Table 5.11.** Model 100 crystals

Crystal	Frequency	Used by
X1	32.768 KHz	Clock/Calendar (chapter 11)
X2	4.9152 MHz	CPU PIO (chapters 7, 9) UART (chapter 7) Beeper (chapter 9)
X3	1.000 MHz	Modem (chapter 8)

In addition to the switches contained in the keyboard, there are five switches controlling major functions of the Model 100. They are listed in table 5.12.

**Table 5.12.** Model 100 switches

Switch	Function
SW-1	Answer/originate switch (chapter 8)
SW-2	Direct/acoustic switch (chapter 8)
SW-3	Memory power switch (chapter 16)
SW-4	Reset pushbutton (chapter 16)
SW-5	On/off switch (chapter 16)





# 6

---

## The Keyboard

---

The Model 100 keyboard has fifty-six conventional typewriter style keys and sixteen small function keys. They are identical electrically. The computer's response to the pressing of a particular key is determined by the program being run. Since most programs use one of two ROM routines for reading the keyboard, it makes sense to think of the keys in terms of the values returned by these routines.

### **Hardware Theory of Operation**

The Model 100's keyboard consists of key switches soldered to a printed circuit board. The keys are numbered on the board and the correspondence between key numbers and the legend printed on the key top is shown in table 6.1. To reorder a key top, you will need the reference number given in the table.

**Table 6.1.** Printed circuit board key designations.

PCB Key	Serv. Ref. no.	Man. Description
1	P-100	f1
2	P-100	f2
3	P-100	f3
4	P-100	f4
5	P-100	f5
6	P-100	f6
7	P-100	f7
8	P-100	f8
9	P-100	PASTE
10	P-100	LABEL
11	P-100	PRINT
12	P-100	BREAK
13	P-100	Leftarrow
14	P-100	Rightarrow
15	P-100	Uparrow
16	P-100	Downarrow
17	P-200	ESC
18	P-101	1
19	P-102	2
20	P-103	3
21	P-104	4
22	P-105	5
23	P-106	■
24	P-107	7
25	P-108	8
26	P-109	9
27	P-110	0
28	P-201	-
29	P-202	=
30	P-203	BKSP
31	P-214	TAB
32	P-127	q
33	P-133	w
34	P-115	e
35	P-128	r
36	P-130	t
37	P-135	y
38	P-131	u
39	P-119	i
40	P-125	o
41	P-126	p

*continued on following page*

PCB Key	Serv. Ref. no.	Man. Description
42	P-204	[
43	P-217	Enter
44	P-215	CTRL
45	P-111	a
46	P-129	s
47	P-114	d
48	P-116	f
49	P-117	g
50	P-118	h
51	P-120	j
52	P-121	k
53	P-122	l
54	P-205	;
55	P-206	'
56	P-207	CAPS
57	P-216	SHIFT
58	P-136	z
59	P-134	x
60	P-113	c
61	P-132	v
62	P-112	b
63	P-124	n
64	P-123	m
65	P-208	,
66	P-209	.
67	P-210	/
68	P-216	SHIFT
69	P-211	GRPH
70	P-218	Space
71	P-212	CODE
72	P-213	NUM

The seventy-two Model 100 keys reside in port space unlike the Model I or III keys which reside in memory address space. This means that in the Models I and III the CPU determines which keys have been pressed by loading in data from certain of the 65536 possible memory addresses. In the Model 100, however, the CPU determines key closures by loading in data from certain of the 256 possible I/O ports. As in the Model I and III, keyboard scanning requires the constant

attention of the CPU. This means that if a period of time passes during which the CPU has not scanned the keyboard, any key pressed during that time is ignored.

Keyboard scanning can occur due to direct action by the program being executed or because certain interrupts are enabled.

## Keyboard Scanning

The electrical layout of the seventy-two keys is an eight-by-nine array, as shown in figure 6.1. Sixty-four keys lie in the main array and are scanned together. They appear in table 6.2. The eight remaining keys, most of which modify other keys, appear in table 6.3.

To determine if a particular key has been pressed, the CPU sends a "0" to a selected bit of output port B9 (or B1; decimal 177 or 185) and "1's" to the other bits. It also makes sure bit 0 of output port BA is on. This selects a column in table 6.2. Then, the byte at input port E8 (any port in the range E0 through EF will do, decimal 224 through 239) is examined. If one of the bits is "0", the key corresponding to that row on table 6.2 has been pressed.

A slightly different procedure applies to the keys in table 6.3. To scan these keys, the CPU sends a "0" to bit 0 of output port BA and all "1's" to output port B9, while examining input port E8. If any of the bits are "0", the key corresponding to that row on table 6.3 has been pressed.

**Table 6.2.** "Main array" key locations in port space. Input takes place through input port E8. If a bit is off, that key has been pressed.

INP E8 bit	Output port B9 bit							
	0	1	2	3	4	5	6	7
7	L	K	I	/	8	Down	ENTER	F8
6	M	J	U	.	7	Up	PRINT	F7
5	N	H	Y	,	6	Right	LABEL	F6
4	B	G	T	'	5	Left	PASTE	F5
3	V	F	R	;	4	=	ESC	F4
2	C	D	E	[	3	-	TAB	F3
1	X	S	W	P	2	0	DEL	F2
0	Z	A	Q	O	1	9	Space	F1

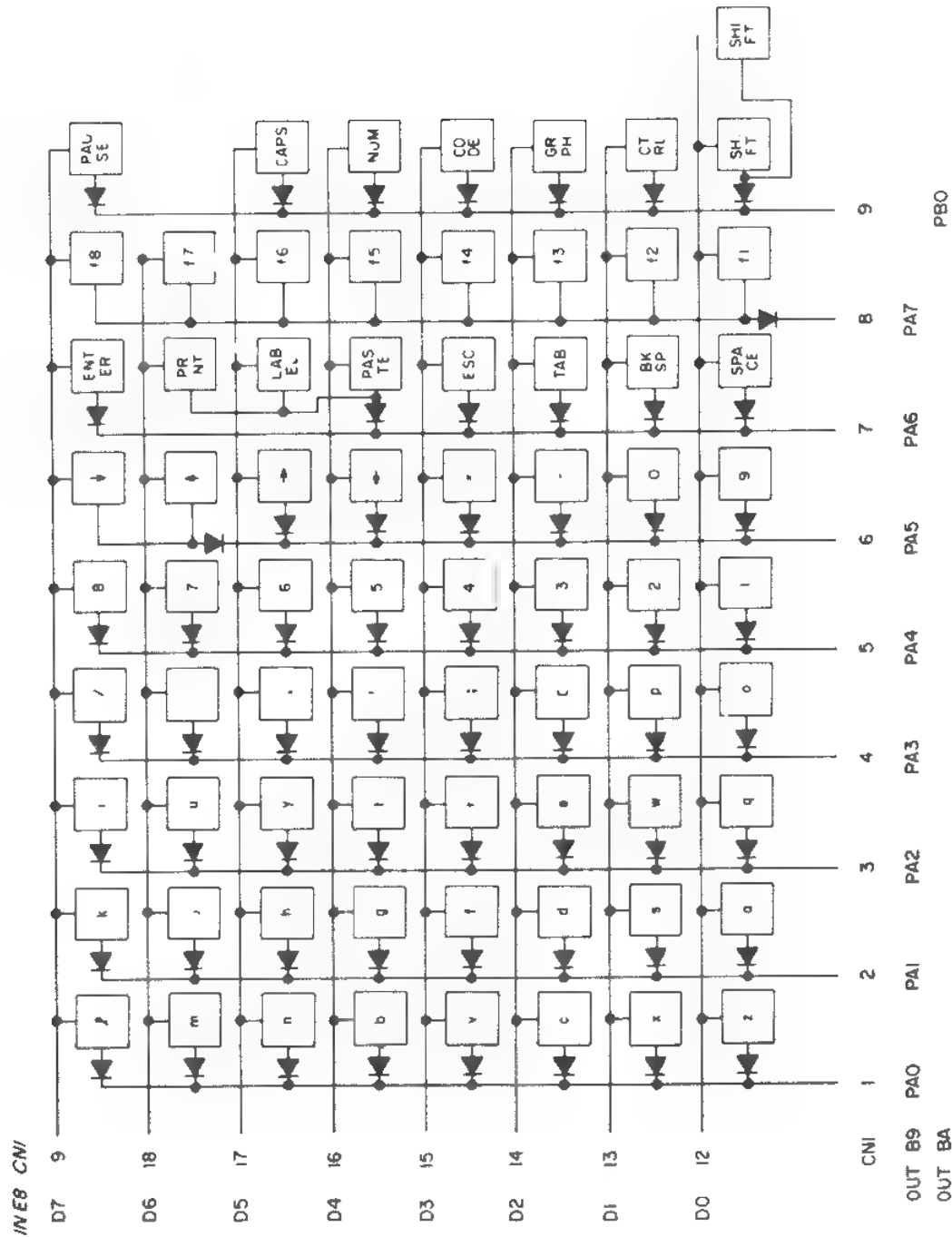


Figure 6.1. Keyboard array

**Table 6.3.** Modifier key locations in port space. Input condition: assumes bit 0 is off at output port BA. Input takes place through input port E8. If a bit is off, that key has been pressed.

Input port BA Bit	Key
7	BREAK
6	Always one (no key at this location)
5	CAPS
4	NUM
3	CODE
2	GRPH
1	CTRL
0	SHIFT (either or both keys)

To see how rows and columns are scanned, enter and run the program shown in figure 6.2.

```

100 FOR I=0 TO 7: OUT 177,255 XOR (2 ^ I):
    A=INP(224): IF A=255 THEN 200
150 PRINT "Table 6.2";I, 255 XOR A
200 NEXT I
201 OUT 178,0 : A=INP(224): IF A=255 THEN 300
202 PRINT "Table 6.3,"255 XOR A
300 GOTO 100

```

**Figure 6.2.** BASIC program demonstrating key scanning.

When you run this program and press a key, the column and row from table 6.2 and table 6.3 are printed. The rows are designated by the numerical value of the bit. For example, 128 means bit 7.

Sometimes all you want to know is whether a key has been pressed. This can be accomplished by sending a "0" to output port B9 and a "0" to bit 0 of output port BA. If input port E8 has the value FF (255), a key has not been pressed. You can see this in figure 6.1. Sending "0's" to the output port sets all nine columns low. If a key had been pressed, then one of the eight data lines at the input port would be low, or logic "0". The numerical value of the input port would be something other than all "1's" (FF, or 255 decimal).

In most circumstances the CAPS, NUM, CODE, GRPH, CTRL, and SHIFT keys are intended to generate a character only in conjunction with a key in the main array. It is often adequate to send all "0's" to output port B9, leaving bit 0 of output port BA at its present "1" state.

### Multiple-Use Ports

Output ports B9 and BA serve many functions besides keyboard scanning. All of the bits of output port B9, for example, are used in keyboard scanning and for line printer output and LCD control. One of the bits is used for serial output to the clock/calendar chip. Bit 0 of output port BA, which is used for keyboard scanning, is also used for LCD control. The other bits of output port BA serve other functions.

Because of the multiple uses of ports, two precautions are in order if you are to do your own keyboard scanning. Avoid interfering with other functions, when using output port BA. Do not disturb bits 1 to 7. This is done by reading in the contents of the output port through input port BA, changing bit 0 through AND and OR statements, and sending the new value out to port BA.

The other precaution is to avoid letting other functions interfere with keyboard scanning. Here is a keyboard input routine that checks if the CTRL-BREAK combination has been pressed:

7D44	3E	EC	MVI A,EC	;LD A,EC
7D46	D3	BA	OUT BA	;OUT (BA),A
7D48	3E	FF	MVI A,FF	;LD A,FF
7D4A	D3	B9	OUT B9	;OUT (B9),A
7D4C	DB	E8	IN E8	;IN A,(E8)
7D4E	E6	82	ANI 82	;AND 82

This is the routine used upon power-up or reset to determine whether CTRL-BREAK was pressed at the moment the power was turned on or at the instant the RESET button was pressed. If so, all RAM files are wiped out and the Model 100 reinitializes all of memory.

The value EC at 7D44 is chosen carefully to avoid turning the power off unintentionally. It activates the modifier column of the



keyboard. The value FF at 7D48 keeps all the keys in the main array out of the scan. The input value received at 7D46 is ANDed with 82 to select bits 7 and 1 of table 6.2. The accumulator is zero only if both the CTRL and BREAK keys have been pressed.

Suppose that an interrupt were to occur between the time of execution of lines 7D46 and 7D4C. The interrupt routine might easily change the contents of output port B9 or BA, allowing keys other than CTRL and BREAK to satisfy the AND 82 test with disastrous results. It is for this reason that interrupts were disabled at line 7D33.

For those who use the published ROM calls for keyboard input, the routines contain the necessary protections to safeguard against a problem arising from the shared use of ports B9 and BA. If you do your own keyboard scanning, however, you should disable or mask interrupts for crucial inputs as discussed in chapter 15.

## **Keyboard ROM Calls**

Radio Shack has published two sets of ROM calls—calls to collect characters from the keyboard and calls to set up the function keys.

Pressing keys provides characters to the Model 100 only if the CPU is paying attention. For keyboard input in the Model 100, it is not necessary for the main program to scan the keyboard repeatedly or even to scan it at all. This is because the CPU is sent a TP (timing pulse) signal from the clock/calendar every four milliseconds. The routine executed by the CPU upon arrival of the TP interrupt includes a keyboard scan. As you will see in chapter 15, the TP interrupt is serviced by the CPU only if interrupts are enabled and if the TP interrupt is unmasked.

When the interrupt routine finds that a key, or more accurately a key-combination has been pressed, the corresponding ASCII code is calculated. There are 128 standardized ASCII values, ranging from zero to 127. The ROM routines return the ASCII value in the A register, which can contain 256 possible values. The Model 100 defines key-combination values for all 256 values. There are 269 key-combinations defined for the Model 100. The additional thirteen possibilities are also returned in the A register but with the carry flag on. These "pseudo-ASCII" values are listed in table 6.4.

**Table 6.4.** Pseudo-ASCII values for certain keys.  
 (Radio Shack publication 700-2245 gives these values incorrectly.  
 This table should be used instead.)

value in	key pressed
accumulator	
0	f1
1	f2
2	f3
3	f4
4	f5
5	f6
6	f7
7	f8
8	LABEL
9	PRINT
A	SHIFT-PRINT
B	PASTE

When keyboard scanning occurs as a result of the TP interrupt, the character returned, if any, is stored in a buffer. This is sometimes called a queue. Keyboard scanning is not the only way characters are loaded into the buffer. Pressing PASTE, or any of the eight function keys, also causes characters to be loaded into the buffer.

If a key is pressed long enough to activate the repeat action, about one second, the ASCII value for that key will show up in the buffer many times.

When characters are loaded into the buffer they do not, in and of themselves, end up on the LCD screen. The main program must arrange for characters to be displayed on the screen if that is desired. Routines to accomplish this are described in chapter 13.

### KEYBOARD INPUT ROM ROUTINES

Perhaps the simplest of the keyboard input routines is one which bears a striking resemblance to the BASIC function INKEY\$. This routine, named KYREAD and called at 7242, determines whether, at the time of the call, any key combination is being pressed.

If no meaningful key combination has been pressed, the routine returns with the Z (zero) flag on. If the Z flag is off, a character has been received, and its value is found in the accumulator. If the C (carry) flag

is on, the value in the accumulator is not an ASCII value but is instead a pseudo-ASCII value and should be interpreted according to table 6.4. This routine is somewhat like the Model I/III routine KBCHAR at 002B.

You may wonder why the routine is defined as responding to "meaningful" key-combinations. Just because a key was pressed at the time of the scan does not mean the result will be located in the accumulator. For example, if GRPH-G is pressed, the routine still comes back with the Z flag on. This is because GRPH-G is not meaningful. Table 6.5 shows this. If you want to be able to detect a GRPH-G, you will have to scan the keyboard yourself, as described earlier in the chapter.

Sometimes you may want to know if a key has been pressed since the CPU last received a character from the keyboard. This can be accomplished by CHSNS, called at 13DB. Upon return from the routine, if the Z flag is set, no keys have been pressed.

Assembly language programmers often wish to assign special significance to two ASCII values, 03 hex and 13 hex, because they have special meanings in BASIC. The value 03 hex, associated with CTRL-C and SHIFT-BREAK, is often used to terminate the program in progress. The value 13 hex, associated with CTRL-S and PAUSE, is often used to temporarily suspend the function in progress, such as scrolling of the screen. A routine is available to determine whether 03 or 13 have been typed. The routine BRKCHK, called at 7283, returns with the C flag set if either character has been received and reset if neither has been received.

The routine KEYX, called at 7270, combines the functions of BRKCHK and CHSNS. If the Z flag is set, no character has been received. If the Z flag is reset, at least one character has been received. If the C flag is set, a CTRL-S or CTRL-C has been received.

If you learn that a character resides in the keyboard buffer, you must obtain the character. This is done by calling CHGET at 12CB. As in the case of KYREAD, the condition of the carry flag indicates whether the value returned in the accumulator is to be understood as ASCII or pseudo-ASCII.

CHGET has one other use. If no character is in the buffer, CHGET causes the computer to wait until a character is typed before returning control to the calling program. In this respect, it is somewhat like the Model I/ II KBWAIT routine at 0049. Put another way, if you do not want to wait for a character, but simply want to process any pending buffer entries, you should check the buffer first (using CHSNS or KEYX) and not call CHGET unless something is there.

A related but unpublished routine is located at 5D64. This routine performs the CHGET function and converts the results to uppercase.

Calling CHGET to retrieve a character from the buffer does not cause the character to be displayed on the screen. The interrupt routine that collects the character and places it in the buffer does not put it on the screen. If you want it to appear on the screen, you have to put it there yourself. This is a complicated task because the entry can be a delete or backspace or another character that requires special attention.

Often you may want to receive a line of characters from the keyboard, terminated by an ENTER (carriage return, ASCII value 0D hex). This could be accomplished with the routines described above but doing so would involve calling routines repeatedly and checking each character to see if it is an ENTER.

The routine INLIN, called at 4644, performs this task. The line that was typed appears in a RAM buffer starting at F685. This routine is somewhat like the Model I/III routine KBLINE at 0040.

As the line is typed, the user is able to correct characters with the left arrow, backspace, or CTRL-H. It is also possible to erase the whole line with CTRL-U and start over. All characters are displayed on the screen. The INLIN routine is used in the BASIC LINE INPUT command (at 0C5F) and is used to collect the user's input in response to the BASIC Ok prompt at 051D, the TELCOM prompt at 516A, the Term Width: prompt at 55D4, and the ADRS and SCHEDL command lines at 5BD2.

## **Label Lines and Functioning Keys**

When a function key (f1 through f8) is pressed, a string of characters associated with that key is loaded to the keyboard buffer. It is possible for the user to change the strings assigned to these function

keys by calling STFNK at 5A7C. Before calling the routine a so-called "f-string" sequence must be set up, and the HL register must point to (contain the address of) the sequence. The sequence must be set up in a precise way; it is composed of eight f-strings, where each f-string is made up of sixteen or fewer characters. If it is made of fewer than sixteen characters, the last character must have bit 7 turned on.

Only the lowest seven bits of the f-string will be displayed on the screen and (when the function key is pushed) loaded to the keyboard buffer. As a result, the CODE and GRAPH characters may not appear in an f-string. (The only exceptions are GRPH-SHIFT-hyphen and GRPH-SHIFT[, with ASCII values 124 and 126 decimal, respectively.)

However, all ASCII values up to 127 decimal may appear in an f-string including such exotic characters as CTRL-G, which appears as the "beep" when sent to the screen.

Let's look at a typical f-string sequence, the TELCOM label line, shown in table 6.5.

**Table 6.5.** TELCOM label line setting

5155	21	51	LXI	HL,51A4;	LD HL,51A4	
5158	CD	7C	5A	CALL	5A7C	
.						
.						
.						
51A4	46	69	6E	64		;"Find"
51A8	A0					;space with bit 7 on
51A9	43	61	6C	6C		;"Call"
51AD	A0					
51AE	53	74	61	74		;"Stat"
51B2	A0					
51B3	54	65	72	6D		;"Term"
51B7	8D					;ENTER with bit 7 on
51B8	80	80	80			;f5, f6, f7 empty
51BB	4D	65	6E	75		;"Menu"
51BF	8D					

This table illustrates a few points about function key labels. If after pushing the function key, the function is to be processed immediately, a command line terminator like ENTER is required. This is the case with function keys f4 and f8.

If, on the other hand, the user can enter characters before pressing ENTER, one may turn on bit 7 of the last label character or append a space with bit 7 on. This the case with function keys f1, f2, and f3.

Finally, if the value 80 is specified for keys that are to be blank, such as keys f5, f6, and f7, the ROM routine CLRFNK at 5F79 loads 80 to all eight keys, resulting in a clearing of all the function keys and labels.

Once the labels are set, other routines cause them to be displayed on the screen — DSPFNK at 42A8, or erased from the screen ERAFNK at 428A.

The actions of STFNK and DSPFNK are combined in STDSPF, at 42A5; this routine, like STFNK, requires that HL be set to point to the f-string sequence.

The RAM location at F63D is used as a label line enable flag. If it is nonzero, the label line is considered to be enabled. (The flag is set at 443B.) The routine FNKSB at 5A9E will cause the function key labels to be displayed (by calling DSPFNK) only if F63D is nonzero.

Function key f-string sequences in ROM include BASIC at 5B46; ADRS and SCHEDL at 5D0A, and 5D1E; TELCOM at 51A4, 5443 and 5D2B; TEXT at 5E15; and a sequence to clear all labels at 5B3E.

## ROM KEYBOARD SCANNING

Several tables are stored in ROM which are used to convert the key closures to ASCII values. Consider first the keys in the first six columns of table 6.1 (except the four arrow keys). These forty-four keys, alone and when modified by the SHIFT, GRPH, and CODE keys, account for the majority of the possible ASCII values returned by INKEY\$ or the ROM calls. The decoding is done with the aid of a table located in ROM at 7BF1 to 7CF8.

For any of these keys, the ASCII value is located at the memory location pointed to by the sum of the following decimal numbers:

- 31729 (start of table)
- bit number 0-7 found to be turned off at input port 232
- eight times the bit number 0-7 turned off at output port 185

- if SHIFT key pressed, add 44
- if GRPH or CODE key pressed, add 88 or 132 respectively.

This ROM table was used to generate table 6.5, which shows the effect of SHFT, GRPH, and CODE keys on the above-mentioned forty-four keys.

The ROM table contains a number of zero entries, e.g. CODE—SHIFT-z. This does not mean that the INKEY\$ value for that combinations of key-presses is CHR\$(0). Instead INKEY\$ returns a null string (""). One way to get a CHR\$(0) is with the input CTRL-SHIFT-2.

Portions of this table are used in several ROM routines, at 718E, 7195, 71A0, 71B2, and 720D.

### **Decoding of Function Keys**

Decoding of keys in the last two columns of table 6.2 is accomplished in similar fashion with ROM tables at 7D0B (for SHIFTEd keys) and 7D10 (for unSHIFTEd keys). To obtain the values returned by ROM call KYREAD or CHGET you must subtract 64 from the value in the ROM table.

### **Decoding the Directional Arrows**

The directional arrows take on differing values depending on whether the SHIFT or CTRL keys are pressed. Table 6.7 shows the values and locations of lookup tables.

**Table 6.6.** ASCII values for SHFT, GRPH, CODE keys

Key	Unshifted	Shift	GRPH	GRPH-Shift	CODE	CODE-Shift
,	39	34	140	0	160	164
'	44	60	153	248	188	221
-	45	95	92	124	197	167
.	46	62	151	247	207	0
/	47	63	138	0	174	0
0	48	41	125	0	175	166
1	49	33	136	225	192	208
2	50	64	156	226	0	0
3	51	35	157	227	193	209
4	52	36	158	228	0	0
5	53	37	159	229	0	0
6	54	94	180	230	0	0
7	55	38	176	0	196	212
8	56	42	163	0	194	210
9	57	40	123	0	195	211
;	59	58	146	245	173	0
=	61	43	141	0	190	168
[	91	93	96	126	181	0
a	97	65	133	235	182	177
█	98	66	149	0	0	0
c	99	67	132	255	162	171
d	100	68	0	237	187	215
e	101	69	143	233	198	214
█	102	70	130	238	0	191
g	103	71	0	253	0	0
h	104	72	134	251	0	0
i	105	73	142	243	199	213
j	106	74	0	244	203	219
k	107	75	155	250	201	217
l	108	76	154	249	202	218
m	109	77	129	246	0	165
█	110	78	150	0	205	0
o	111	79	152	242	183	178
p	112	80	128	241	172	0
q	113	81	147	231	200	216
r	114	82	137	234	0	170
s	115	83	139	236	169	185
t	116	84	135	252	0	186
█	117	85	145	240	184	179
v	118	86	0	0	189	222
w	119	87	148	232	0	0
x	120	88	131	239	161	223
y	121	89	144	254	204	220
█	122	90	0	224	206	0



**Table 6.7.** ASCII values for Arrow keys

	Left	Right	Up	Down	ROM Table Location
Unshifted	29	28	30	31	7D1B
SHIFT	1	6	20	2	7D07
CTRL	17	18	23	26	7D2F*

## Nums Decoding

Certain keys take on alternate meanings when the NUMS key is down. The routine to convert such keys is located at 7233-7241, and it uses a table at 7CF9-7D06. Assuming the key pressed has already been decoded to ASCII, the lower-case value is compared to the value at 7CF9, 7CFB, and so. If there is a match, the value immediately following (7CFA, 7CFC, etc.) is substituted.

## Arriving At A Particular INKEY\$ Value

The question often arises what combination of keystrokes will produce a certain INKEY\$ value. The ASCII table given in the Radio Shack manuals is incomplete and incorrect. The correct and complete repertoire of keystrokes is given in table 6.8. Note, for example, that in some cases (e.g. 1, 2, 6, 8, 9, 13, 17, 18, 20, 23, and 26) more than one combination of keystrokes will produce a given INKEY\$ value.

---

\* Subtract 64 from table values.

**Table 6.8.** INKEY\$ keystroke combinations

Decimal	Hex	LCD	INKEY\$
1	01		CTRL-a      SHIFT-Leftarrow
2	02		CTRL-b      SHIFT-Downarrow
3	03		CTRL-c
4	04		CTRL-d
5	05		CTRL-e
6	06		CTRL-f      SHIFT-Rightarrow
7	07		CTRL-g
8	08		CTRL-h      BKSP
9	09		CTRL-i      TAB
10	0A		CTRL-j
11	0B		CTRL-k
12	0C		CTRL-l
13	0D		CTRL-m      ENTER
14	0E		CTRL-n
15	0F		CTRL-o
16	10		CTRL-p
17	11		CTRL-q      CTRL-Leftarrow
18	12		CTRL-r      CTRL-Rightarrow
19	13		CTRL-s
20	14		CTRL-t      SHIFT-Uparrow
21	15		CTRL-u
22	16		CTRL-v
23	17		CTRL-w      CTRL-Uparrow
24	18		CTRL-x
25	19		CTRL-y
26	1A		CTRL-z      CTRL-Downarrow
27	1B		ESC
28	1C		Rightarrow
29	1D		Leftarrow
30	1E		Uparrow
31	1F		Downarrow
32	20		Space
33	21	!	SHIFT-1
34	22	"	SHIFT-'
35	23	#	SHIFT-3
36	24	\$	SHIFT-4
37	25	%	SHIFT-5

Decimal	Hex	LCD	INKEY\$
38	26	&	SHIFT-7
39	27	/	'
40	28	<	SHIFT-9
41	29	>	SHIFT-0
42	2A	*	SHIFT-8
43	2B	+	SHIFT-=
44	2C	,	,
45	2D	-	-
46	2E	.	.
47	2F	/	/
48	30	0	0 NUM-m
49	31	1	1 NUM-j
50	32	2	2 NUM-k
51	33	3	3 NUM-l
52	34	4	4 NUM-u
53	35	5	5 NUM-i
54	36	6	6 NUM-o
55	37	7	7
56	38	8	8
57	39	9	9
58	3A	:	SHIFT-;
59	3B	;	:
60	3C	<	SHIFT-,
61	3D	=	=
62	3E	>	SHIFT-.
63	3F	?	SHIFT-/
64	40	@	SHIFT-2
65	41	A	SHIFT-a
66	42	B	SHIFT-b
67	43	C	SHIFT-c
68	44	D	SHIFT-d
69	45	E	SHIFT-e
70	46	F	SHIFT-f
71	47	G	SHIFT-g
72	48	H	SHIFT-h
73	49	I	SHIFT-~
74	4A	J	SHIFT-j
75	4B	K	SHIFT-k

Decimal	Hex	LCD	INKEY\$
76	4C	L	SHIFT-l
77	4D	M	SHIFT-m
78	4E	N	SHIFT-n
79	4F	O	SHIFT-o
80	50	P	SHIFT-p
81	51	Q	SHIFT-q
82	52	R	SHIFT-r
83	53	S	SHIFT-s
84	54	T	SHIFT-t
85	55	U	SHIFT-u
86	56	V	SHIFT-v
87	57	W	SHIFT-w
88	58	X	SHIFT-x
89	59	Y	SHIFT-y
90	5A	Z	SHIFT-z
91	5B	[	[
92	5C	\	GRPH--
93	5D	]	SHIFT-[
94	5E	^	SHIFT-6
95	5F	_	SHIFT--
96	60	`	GRPH-[
97	61	a	a
98	62	b	b
99	63	c	c
100	64	d	d
101	65	e	e
102	66	f	f
103	67	g	g
104	68	h	h
105	69	i	i
106	6A	j	j
107	6B	k	k
108	6C	l	l
109	6D	m	m
110	6E	n	n
111	6F	o	o
112	70	p	p
113	71	q	q

Decimal	Hex	LCD	INKEY\$
114	72	r	r
115	73	s	s
116	74	t	t
117	75	u	u
118	76	v	v
119	77	w	w
120	78	x	x
121	79	y	y
122	7A	z	z
123	7B	(	GRPH-9
124	7C		GRPH-SHIFT--
125	7D	)	GRPH-0
126	7E	~	GRPH-SHIFT-[
127	7F		SHIFT-BKSP
128	80	8	GRPH-p
129	81	9	GRPH-m
130	82	h	GRPH-f
131	83	e	GRPH-x
132	84	4	GRPH-c
133	85	+	GRPH-a
134	86	h	GRPH-h
135	87	t	GRPH-t
136	88	i	GRPH-l
137	89	r	GRPH-r
138	8A	/	GRPH-/
139	8B	s	GRPH-s
140	8C	≈	GRPH-'
141	8D	±	GRPH-=
142	8E	j	GRPH-i
143	8F	4	GRPH-e
144	90	h	GRPH-y
145	91	h	GRPH-u
146	92	4	GRPH-;
147	93	4	GRPH-q
148	94	4	GRPH-w
149	95	4	GRPH-b
150	96	4	GRPH-n
151	97	4	GRPH-.
152	98	↑	GRPH-o

Decimal	Hex	LCD	INKEY\$
153	99	↓	GRPH-,
154	9A	→	GRPH-1
155	9B	←	GRPH-k
156	9C	⊕	GRPH-2
157	9D	⊖	GRPH-3
158	9E	♥	GRPH-4
159	9F	⊙	GRPH-5
160	A0	‘	CODE-’
161	A1	Ⓐ	CODE-x
162	A2	Ⓢ	CODE-c
163	A3	Ⓔ	GRPH-8
164	A4	`	CODE-SHIFT-’
165	A5	μ	CODE-SHIFT-m
166	A6	+	CODE-SHIFT-0
167	A7	▼	CODE-SHIFT--
168	A8	↑	CODE-SHIFT-=
169	A9	Ⓔ	CODE-s
170	AA	Ⓢ	CODE-SHIFT-r
171	AB	Ⓢ	CODE-SHIFT-c
172	AC	Ⓢ	CODE-p
173	AD	Ⓢ	CODE-;
174	AE	Ⓢ	CODE-;/
175	AF	Ⓢ	CODE-0
176	B0	Ⓢ	GRPH-7
177	B1	Ⓐ	CODE-SHIFT-a
178	B2	Ⓢ	CODE-SHIFT-o
179	B3	Ⓢ	CODE-SHIFT-u
180	B4	Ⓢ	GRPH-6
181	B5	~	CODE-[
182	B6	Ⓢ	CODE-a
183	B7	Ⓢ	CODE-o
184	B8	Ⓢ	CODE-u
185	B9	Ⓢ	CODE-SHIFT-s
186	BA	Ⓢ	CODE-SHIFT-t
187	BB	Ⓢ	CODE-d
188	BC	Ⓢ	CODE-,
189	BD	Ⓢ	CODE-v
190	BE	..	CODE-=
191	BF	Ⓢ	CODE-SHIFT-f

Decimal	Hex	LCD	INKEY\$
192	C0	â	CODE-1
193	C1	ê	CODE-3
194	C2	î	CODE-8
195	C3	ô	CODE-9
196	C4	û	CODE-7
197	C5	^	CODE--
198	C6	ë	CODE-e
199	C7	ï	CODE-i
200	C8	ä	CODE-q
201	C9	í	CODE-k
202	CA	ó	CODE-l
203	CB	ú	CODE-j
204	CC	ý	CODE-y
205	CD	ñ	CODE-n
206	CE	ã	CODE-z
207	CF	ö	CODE-.
208	D0	â	CODE-SHIFT-1
209	D1	ê	CODE-SHIFT-3
210	D2	î	CODE-SHIFT-8
211	D3	ô	CODE-SHIFT-9
212	D4	û	CODE-SHIFT-7
213	D5	ï	CODE-SHIFT-i
214	D6	ë	CODE-SHIFT-e
215	D7	é	CODE-SHIFT-d
216	D8	ä	CODE-SHIFT-q
217	D9	í	CODE-SHIFT-k
218	DA	ó	CODE-SHIFT-l
219	DB	ú	CODE-SHIFT-j
220	DC	ý	CODE-SHIFT-y
221	DD	û	CODE-SHIFT-,
222	DE	ë	CODE-SHIFT-v
223	DF	à	CODE-SHIFT-x
224	E0		GRPH-SHIFT-z
225	E1	▪	GRPH-SHIFT-1
226	E2	•	GRPH-SHIFT-2
227	E3	▪	GRPH-SHIFT-3
228	E4	▪	GRPH-SHIFT-4
229	E5	▪	GRPH-SHIFT-5

Decimal	Hex	LCD	INKEY\$
230	E6	⌞	GRPH-SHIFT-6
231	E7	⌞	GRPH-SHIFT-q
232	E8	⌞	GRPH-SHIFT-w
233	E9	⌞	GRPH-SHIFT-e
234	EA	⌞	GRPH-SHIFT-r
235	EB	⌞	GRPH-SHIFT-a
236	EC	⌞	GRPH-SHIFT-s
237	ED	⌞	GRPH-SHIFT-d
238	EE	⌞	GRPH-SHIFT-f
239	EF	⌞	GRPH-SHIFT-x
240	F0	⌞	GRPH-SHIFT-u
241	F1	⌞	GRPH-SHIFT-p
242	F2	⌞	GRPH-SHIFT-o
243	F3	⌞	GRPH-SHIFT-l
244	F4	⌞	GRPH-SHIFT-j
245	F5	⌞	GRPH-SHIFT-i
246	F6	⌞	GRPH-SHIFT-m
247	F7	⌞	GRPH-SHIFT-.
248	F8	⌞	GRPH-SHIFT-,
249	F9	⌞	GRPH-SHIFT-1
250	FA	⌞	GRPH-SHIFT-k
251	FB	⌞	GRPH-SHIFT-h
252	FC	⌞	GRPH-SHIFT-t
253	FD	⌞	GRPH-SHIFT-g
254	FE	⌞	GRPH-SHIFT-y
255	FF	⌞	GRPH-SHIFT-c





# 7

---

## UART Operation And The RS-232 Interface

---

This chapter examines the UART, a specialized integrated circuit used for serial communications. The RS-232 port is also discussed in detail. The UART is used also for modem communications, a topic covered in chapter 8.

### **Parallel and Serial Data**

Most communication within the Model 100 and with outside devices is accomplished using parallel data busses. *Parallel bus* refers to a group of data paths (usually wires), which at a given instant convey several bits of data (usually eight), each representing a one or a zero.

A serial transmission line, on the other hand, involves a single data path, which at a given instant conveys a single bit of data.

All other factors being equal, a parallel bus conveys data faster than a serial bus and requires less hardware per bit.

Nonetheless, serial transmission circuitry is an essential part of any computer, for two major reasons. First, any data to be transmitted by audio signals, e.g. by telephone or cassette, can only be sent serially since the medium permits only a limited number of states to represent data values. With 300-baud modem communication, for example, one audio tone represents a “1” and another tone represents a “0”.

Secondly, a serial interface was required in the Model 100 to make it compatible with the many computers and peripheral devices that have serial interfaces intended to meet the RS-232C standard.

In the Model 100, a universal asynchronous receiver/transmitter, or UART, is used to convert the Model 100’s parallel data to serial data. A multiplexer is used to connect the UART either to the RS-232C port or to the telephone modem circuitry.

### CONVERTING FROM PARALLEL DATA TO SERIAL DATA

The process of parallel-to-serial conversion is shown in Figure 7.1. Figure 7.1a depicts an eight-bit binary word being sent down a serial transmission line. (The term *word* is used in this chapter as synonymous with the term *eight-bit byte*.) The fourth bit is just leaving the TR (transmitter register) at the time depicted here. (The fact that a single 0 or 1 state is in transition here is the defining characteristic of a so-called serial bus.)

In figure 7.1b, we see that another word has been loaded into the TR from the TBR (transmitter buffer register). Not shown in this figure is the means by which the transmitter circuitry informs the CPU that the TBR is now ready to accept another word.

In figure 7.1c we see, in transition, the loading of yet another word into the TBR. At this instant eight possible 0’s or 1’s are on the verge of being communicated to the TBR. (This is the defining characteristic of a so-called *parallel bus*.)

### What Is A UART?

UART is an acronym for *universal asynchronous receiver/transmitter*. The parallel-to-serial process just described is termed the *transmitter* function. The reception of serial data from a distant device and conversion to parallel data is termed the *receiver* function. The

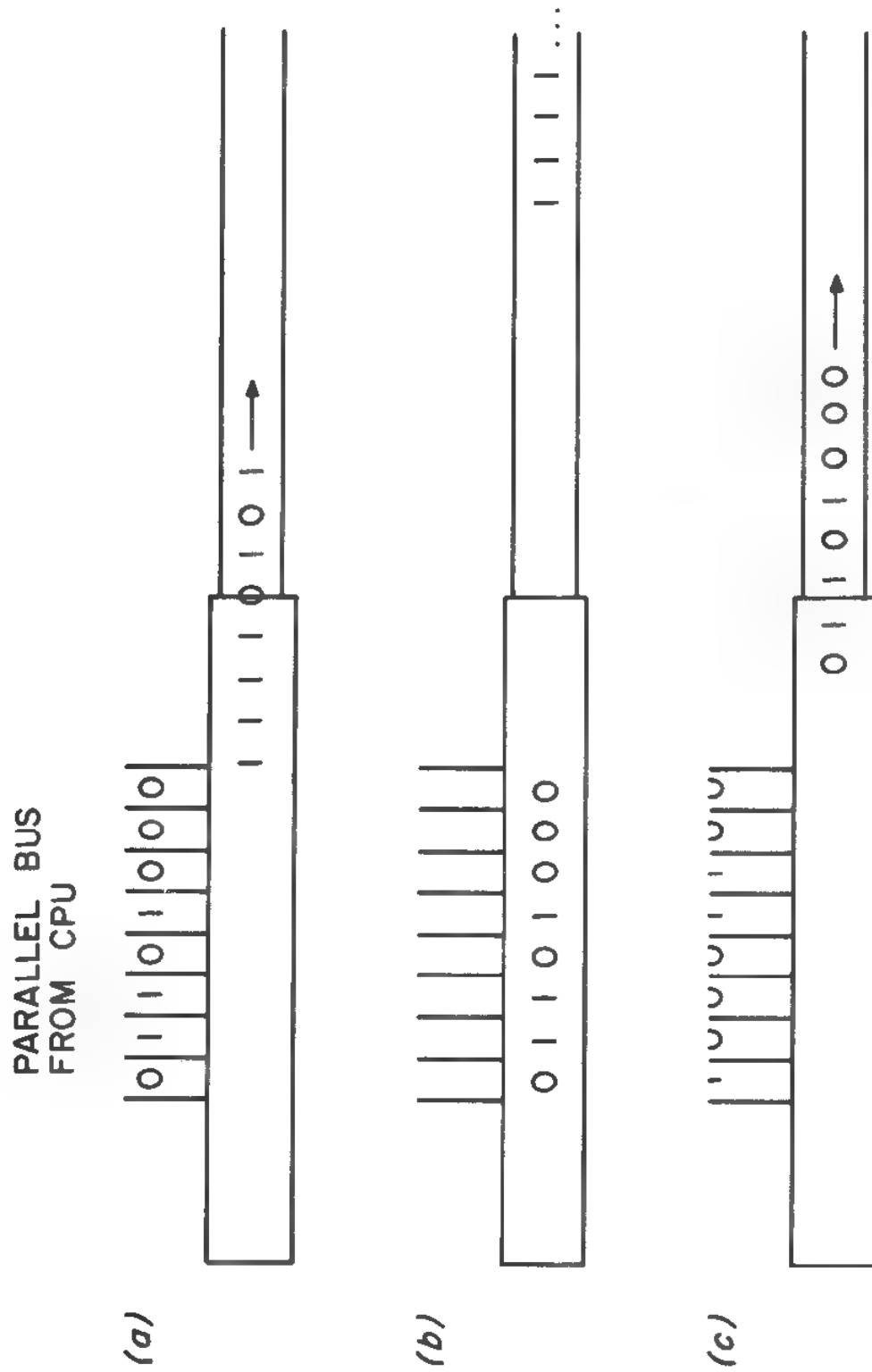


Figure 7.1. Parallel to serial conversion

adjective *asynchronous* means that the receiver can handle incoming words at irregular intervals. There is no requirement that subsequent bursts of 0's and 1's be separated by an unchanging interval, nor that any warning (other than a so-called *start bit*) be given that a word is forthcoming. It is termed *universal* because it is capable of being programmed for any of a variety of word lengths, data rates, and so on.

As with virtually all Model 100 integrated circuits, information enters and leaves the UART on conductors which carry 5 volts to represent a 1 and 0 volts to represent a 0. Thus the eight 1's and 0's depicted schematically in figure 7.1c are in reality eight voltage levels on eight wires. (The wires are pins 26-33 of the integrated circuit.) Similarly, the 1's and 0's shown leaving to the transmitter to the right represent periods of time during which a wire (pin 25) carried either a 5-volt signal or a 0-volt signal.

### TIMING

Suppose a log was made of the voltage at the output of the transmitter as time passes. What would be observed? The result for transmission of a capital "A" at 300 baud is shown in table 7.1.

**Table 7.1.** Transmitter output voltage as a function of time (transmission of uppercase A at 300 baud)

<b>Before Transmission</b>	<b>5 volts</b>	
<b>Transmission Begins</b>		
(set $t=0$ )	0 volts	Beginning of start bit
$t= 3.3$ msec	5 volts	Beginning of least significant bit (bit 0)
$t= 6.6$ msec	0 volts	Beginning of next bit(bit 1)
$t= 23.3$ msec	5 volts	(bit 6)
$t= 26.6$ msec	0 volts	Most significant bit (bit 7)
$t=30$ msec (and thereafter)	5 volts	Stop bit or bits

Prior to the transmission of a character, the UART is putting out a five volt signal. The UART signals that it will soon be transmitting a word by dropping the voltage to 0 for one *bit time*. In the example shown the bit time is about 3.33 milliseconds, or 300 bits per second.

(The number of bits per second is called the *baud rate* after J.M.E. Baudot, a French inventor who died in 1903 and pioneered in the field of serial communications.)

The next eight bit times vary between five volts or zero volts corresponding to the bits of the word being transmitted. For example, the numerical value of a capital A according to the ASCII standard code is 65, or 01000001 in binary. The bits are labeled bit 7, bit 6, and so on, down to bit 0 at the right end of the binary number. By convention, the least significant bit, bit 0, is transmitted first. Thus, for one bit time, the output is again at five volts. Then, for about 16.6 milliseconds, the transmitter puts out zero volts. This is because bits 1 to 5 of the ASCII "A" are zero. (The 16.6 millisecond period is composed of five *windows*, each 3.3 milliseconds long. These might have been five volts had some character other than "A" been transmitted.)

The "1" of bit 6 is then represented by five volts for 3.3 milliseconds followed by the 0 of bit 7, which appears as zero volts for 3.3 milliseconds.

A parity bit would appear at this point in the sequence if one was being sent. With even parity, the parity bit is selected so as to make an even number of 1's in the transmitted word; with odd parity, it is chosen to make an odd number of 1's.

Finally, one or more stop bits, composed of a logic 1 (a 5 volt output) are sent. The stop bits are indistinguishable from the period of logic 1 (5 volts) that lies between the time the stop bit of this word has finished and the time the next word begins.

## The Inner Workings of the UART

The architecture of the UART is shown in figure 7.2. It is a forty-pin integrated circuit, type number IM6402, and is designated M22 in the Model 100. The transmit and receive rates are determined by the TRC and RRC (transmitter register clock and receiver register clock) signals entering the UART at the left.

Parallel data enter at the top, passing through the TBR into the TR, through a multiplexer and then out the TRO (transmitter register output) line.

Received data enter by way of the RRI (receiver register input) line, through a multiplexer to the RR (receiver register), thence to the RBR (receiver buffer register).

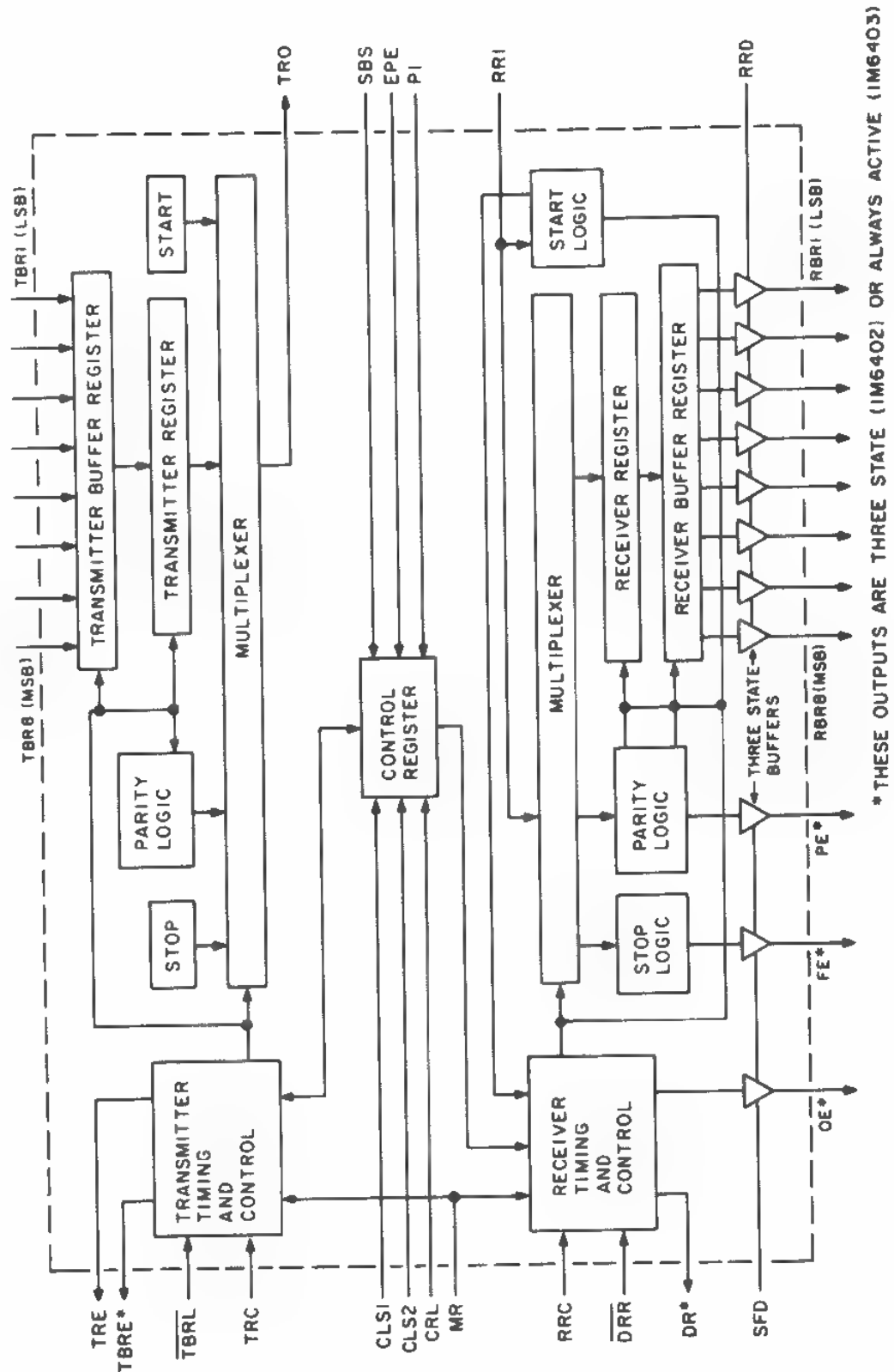


Figure 7.2. UART architecture

The UART requires certain one-wire signals for word length (CLS1 and CLS2), stop bit selection (SBS), and parity (PI, EPE). These signals are loaded (through CPU ports) as discussed in the following section.

## CPU Communication with the UART

The CPU tells the UART how to configure the transmitted words (as well as what sort of words to expect to receive), provides a baud rate frequency for the UART, gives the UART characters to transmit, responds to the UART's signal that a character has been received, and learns whether any errors occurred in data reception. Most of these functions take place through the CPU input and output ports; one function occurs by way of a CPU interrupt. Each of these processes will be discussed in turn.

### SERIAL WORD CONFIGURATION

Before serial I/O can take place, the CPU informs the UART how the transmitted words should be formed and what sort of words to expect to receive.

The UART parameters, namely word length, parity, and stop bit selection, are all loaded by the CPU through output port D8, which is listed in table 7.2 and shown in figure 7.3. Actually, any port number in the range D0 (decimal 208) to DF (223) will do.

**Table 7.2.** UART and other signals (output port D8)

Bit	Function
0	SBS (0=one stop bit; 1=two stop bits*)
1	EPE (0=odd parity; 1=even parity; ignored by UART if bit 2 is on.)
2	PI (1=no parity)
3	CLS1 (0=5 or 7 bits; 1=6 or 8 bits)
4	CLS2 (0=5 or 6 bits; 1=7 or 8 bits)
5	not used
6	not used
7	not used

\* If the character length is set to 5, SBS=1 yields 1.5 stop bits.



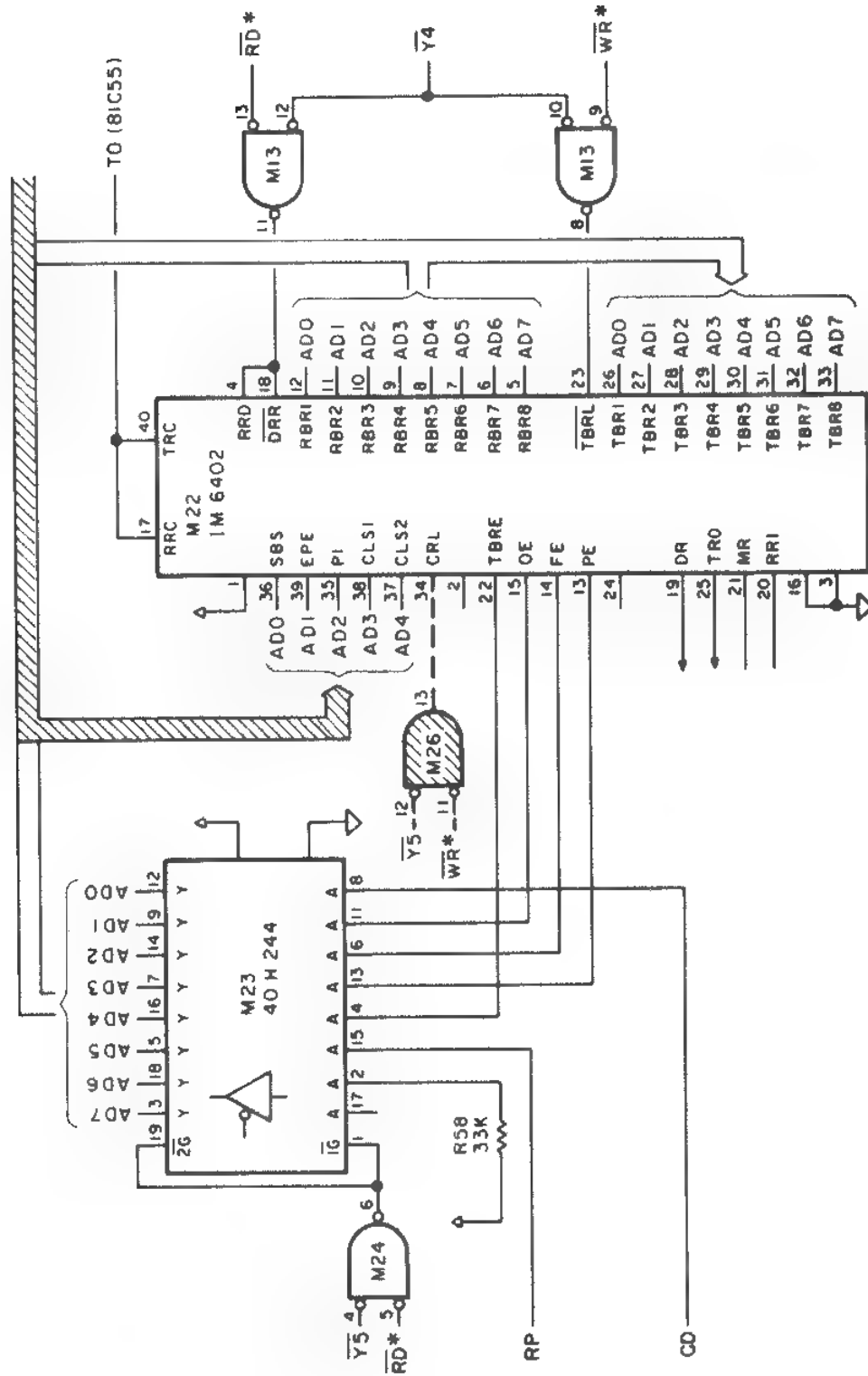


Figure 7.3. UART programming

### SETTING THE BAUD RATE

CPU selection of baud rate is accomplished by loading a divisor into the PIO timer register through output ports BC or B4 (decimal 180 or 188) and BD or B5 (decimal 181 or 189).

Baud rate depends ultimately upon crystal X2, located near the CPU, which oscillates at 4.9152 megahertz. This is shown in photo 7.1. The CPU provides a CLK signal of 2.4576 megahertz (half the crystal frequency) to the PIO at its TIMER IN pin. The PIO divides the TIMER IN frequency by the divisor previously loaded into the PIO timer register. It then provides that reduced frequency via the TIMER OUT pin to the UART at both the TRC and RRC pins. (The PIO also provides the TIMER OUT signal to the piezoelectric beeper for sound generation). The UART in turn divides the PIO signal by 16 to arrive at both the transmit and receive baud rates. The divisors necessary to produce commonly used baud rates are shown in table 7.3. Note that while most of the baud rates are exact, the PIO output for 110 baud is in error by about 0.026%. A divisor of 1396 yields a TRC/ RRC value of 1760.4585 hertz, which results in a data rate of 110.02865 baud.

**Table 7.3.** Baud rate divisors (decimal).

Baud Rate	PIO Divisor	Upper Byte Port BD	Lower Byte Port BC	TELCOM Serial Value
75	2048	72	0	1
110	1396	69	116	2
300	512	66	0	3
300	512	66	0	M
600	256	65	0	4
1200	128	64	128	5
2400	64	64	64	6
4800	32	64	32	7
9600	16	64	16	8
19200	8	64	8	9

Note that each value sent to output port BD has bit 6 on and bit 7 off. This is because port BD actually performs two functions. Bits 0 through 5 are the upper byte of the divisor, while bits 6 and 7 determine

the divider mode. As mentioned in chapter 5, the timer may be set by means of bits 6 and 7 for a single cycle of square wave (00), a single pulse (10), a continuous square wave (01), or continuous pulses (11). The UART requires a continuously provided frequency, so only the latter two modes work properly. To avoid any danger of the pulse being too brief for the UART to pick it up, the safest thing is to use the square wave. Thus, the byte sent to port BD has bit 7 off and bit 6 on.

One more CPU action, an output to port B8, is needed to provide the data rate clock signal to the UART. The PIO divisor must be enabled (port value C3) rather than disabled (port value 43).

### **SERIAL TRANSMISSION**

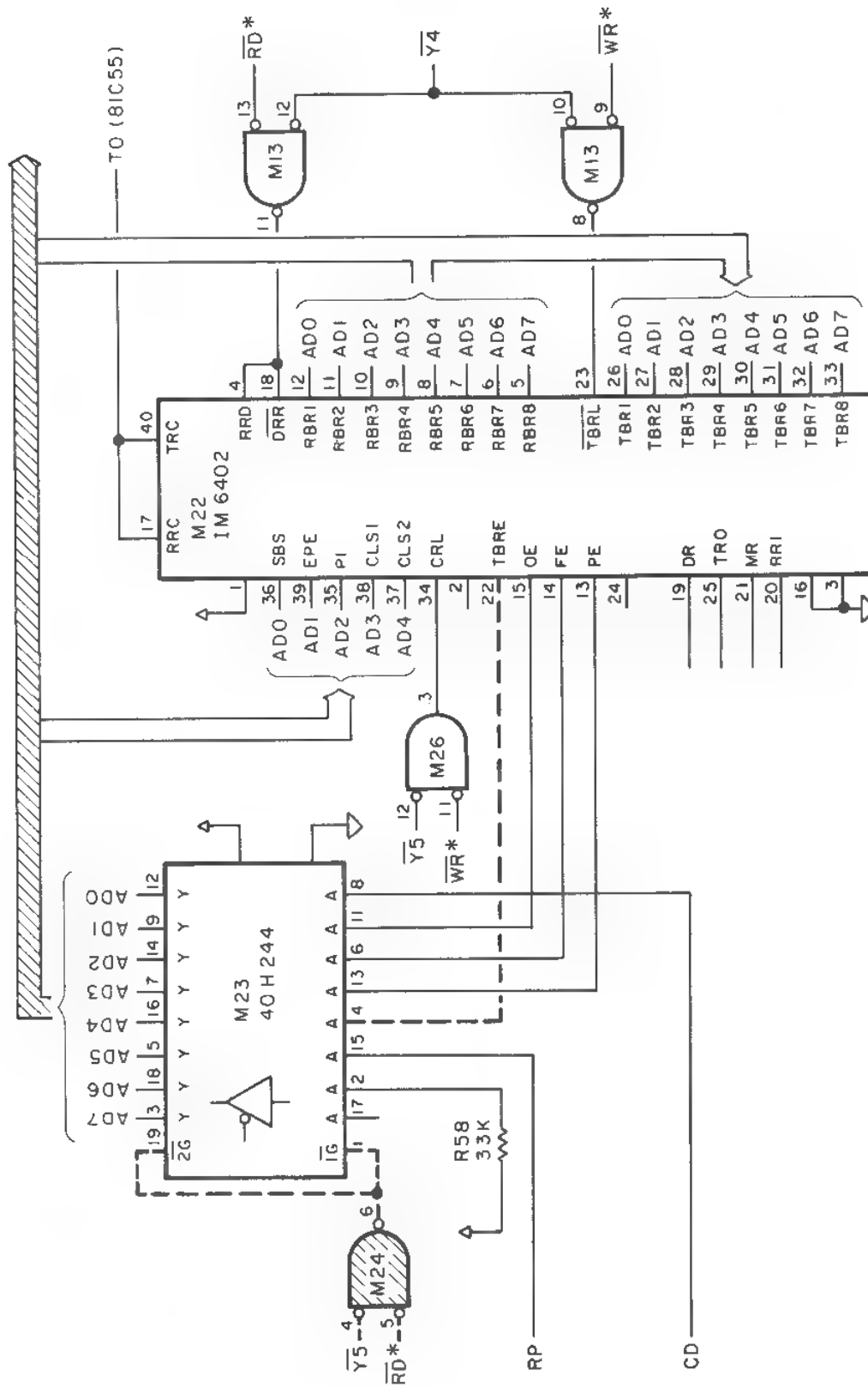
Before sending a character by way of the UART, the CPU must confirm that any character in the TBR has already been received by the TR. The CPU does this by inspecting the condition of bit 4 of input port D8 (or D0 decimal 208 or 216). That bit carries the UART signal TBRE (transmitter buffer register empty) signal (*see table 7.4*). This is shown schematically in figure 7.4a.

The UART is designed so that once a character resides in the TR (and will presumably be transmitted presently), the CPU can load another character into the TBR. Presumably a certain (nonzero) amount of time passes between the moment the "empty" signal is sent to the CPU and the moment the next character is provided to the UART. The presence of the TBR means that the UART can, upon finishing the sending character, have another one ready to send which is "waiting in the wings."

Thus far nothing has been said about how the CPU loads the outgoing data into the UART. This is accomplished through output port C8 (or C0; decimal 192 or 200). The process is shown schematically in figure 7.4b.

### **SERIAL DATA RECEPTION**

When an entire character has been received and loaded to the RBR (receiver buffer register), the UART indicates this by producing a logic "1" at its DR (data received) pin. In the Model 100, this is wired to the RST 6.5 interrupt pin of the CPU. As is discussed in detail in



**Figure 7.4a.** UART data transmission



**Figure 7.4b.** UART data transmission

chapter 15, this causes a subroutine call to ROM location 0034, which disables interrupts and jumps to 6DAC. The CPU obtains the received byte through input port C8 (or C0 decimal 192 or 200). There, the CPU places the received character in a RAM buffer, as shown in figure 7.5.

As is discussed in chapter 15, the usual ROM handling of a received character can be circumvented by disabling interrupts or by changing a RAM vector at F5FC.

### DATA RECEPTION ERRORS

Any number of things can go wrong in serial reception of data:

- The CPU could take too long to pick up a received character
- A "0" might occur when a stop bit one was expected
- The number of 1's in the received byte might differ from that indicated by the parity bit.

Such errors are known as *overflow errors* (OE), *framing errors* (FE), and *parity errors* (PE), respectively. These signals are made available to the CPU at input port D8 (or D0 decimal 208 or 216); the connection is shown in figure 7.4a. The various bits of the input port are shown in table 7.4. To obtain the UART error, the value returned at the port should be ANDed with 0E.

When a receive error has occurred, the received value at input port C8 may or may not be correct.

**Table 7.4.** UART and other signals (input port D8) (the information contained in bit 0 varies depending on the RS-232 modem mode)

Bit	In RS-232C Mode	In Modem Mode
0	Always 0	1=carrier detect
1	OE (1=UART overrun error)	same
2	FE (1=UART framing error)	same
3	PE (1=UART parity error)	same
4	TBRE (1=UART transmitter Buffer register empty)	same
5	RP (0=ground at PHONE pin 8)	same

Continued on following page

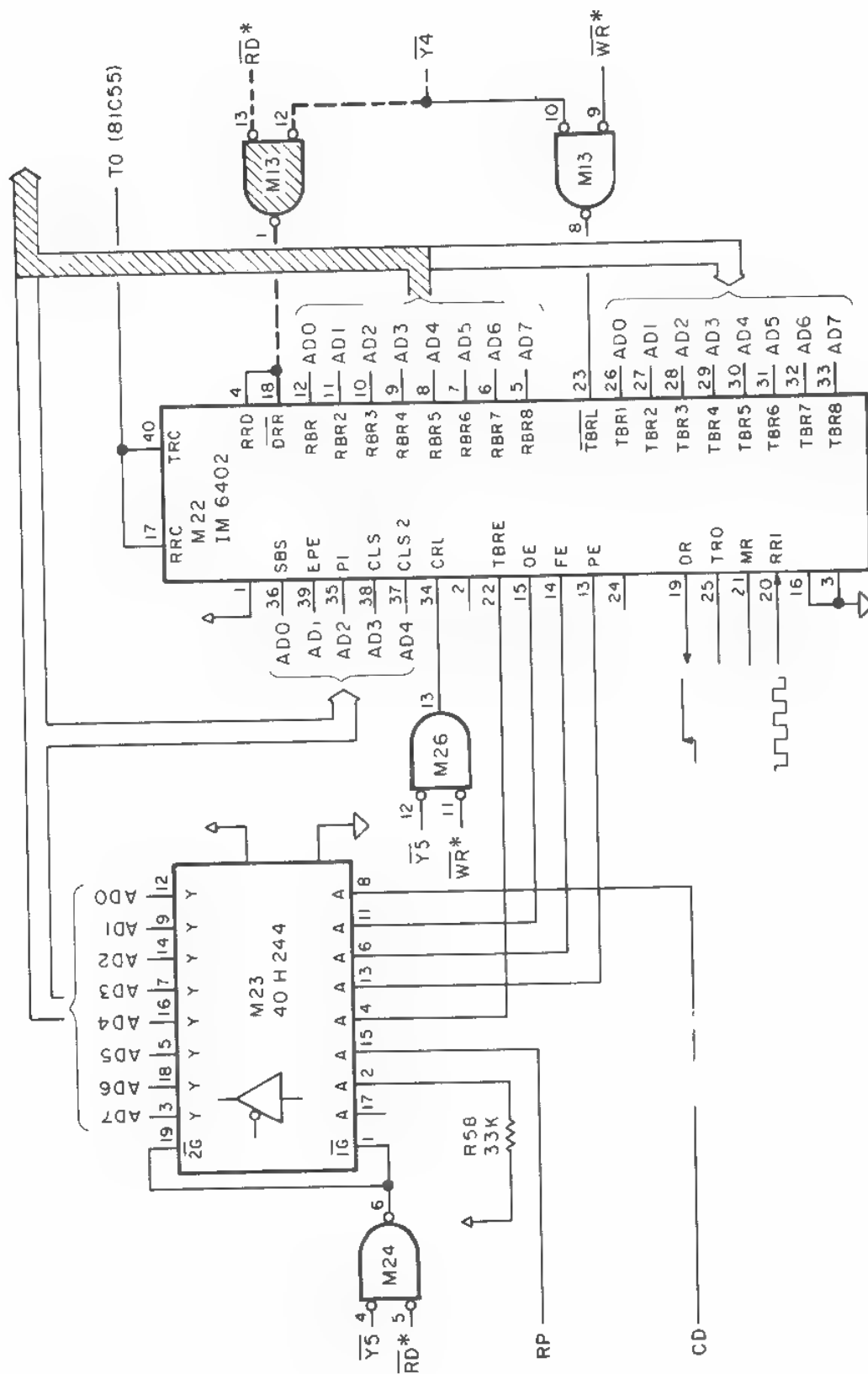
Bit	In RS-232C Mode	In Modem Mode
6	Hobby use (0=ground at M23 pin 2)	same
7	Low-power signal (0= battery low)	same

### Significance of ASCII Code

Nothing in the UART hardware, nor in the ROM routines relating to the UART, requires that the information sent or received be coded according to the ASCII (American Standard Code for Information Interchange) standard. In plain language there is no reason that the value 32 (decimal) must represent a space nor that the value 65 (decimal) represent an uppercase A. You can devise your own coding system if you desire using the Model 100 hardware to communicate. For example, many teletype machines use a five-bit Baudot code and many IBM machines use an EBCDIC code (Extended Binary Coded Decimal Interchange Code), each of which assigns values different from ASCII to the various letters of the alphabet, numerals, and so on.

You are perfectly free within software to translate between ASCII (used in the Model 100 keyboard and display screen) and any other code used by a distant device.

The only exception to this is that in the ROM UART routines, the values 17 and 19 decimal are treated differently than the other 254 values, due to their ASCII-assigned meanings as XON and XOFF. Many problems with XON XOFF can be avoided simply by disabling XON XOFF, or one can write UART routines (borrowing from the ROM routines) that are neutral regarding the values 17 and 19.



**Figure 7.5.** Received UART data



## The Beeper

The BEEP routine, which can be invoked by sending an ASCII 7 to the screen, works fine, regardless of what the baud rate divider is doing. As a result, an ASCII BELL sent by the remote device beeps at the Model 100. (This is explained in detail in chapter 9.)

## ASCII Protocol — XON/XOFF

If XON/XOFF is enabled, the Model 100 will monitor the incoming stream of characters for a CTRL-S. If a CTRL-S is received, the computer will delay sending any characters until such time as a CTRL-Q is received. Three flags are kept relating to XON/XOFF: FF40 indicates whether XON/XOFF was enabled (nonzero) or disabled (zero) during the Stat initialization, FF41 indicates whether the Model 100 most recently transmitted a CTRL-S (nonzero) or a CTRL-Q (zero), and FF42 indicates whether the other device most recently sent the Model 100 a CTRL-S (nonzero) or CTRL-Q (zero).

## Mode Selection: RS-232 and Modem

The UART serves as the serial-to-parallel and parallel-to-serial convertor for the RS-232 and telephone modem interfaces. A device termed a *multiplexer* connects the UART either to the RS-232 interface or to the telephone modem interface. The multiplexer will be discussed here. Then, with the assumption that the multiplexer is set in RS-232 mode, the balance of the chapter will be devoted to the RS-232 interface. The modem mode is discussed in chapter 8.

The UART multiplexer is a handful of components set up to act like a big six-pole, double-throw switch. The position of the switch is determined by bit 3 of output port BA. Since that port controls many other functions including power control (*see* chapter 5), one must be careful how port BA is set. A safe setting for selecting the modem is 2D hex, and a safe setting for RS-232 mode is 25 hex. (*See* for example, the ROM code at 6EAA through 6EB8, in which these values are used.) When the computer is powered up, the multiplexer is in *modem* mode. A change to RS-232 mode is caused by any of the following: setting Stat to a baud rate other than M, opening COM: as a file in BASIC, or loading 0 to bit 3 of output port BA directly.

The values switched by the multiplexer are detailed in table 7.5.

**Table 7.5.** Multiplexer configuration

Signal	RS-232 mode	Modem mode
RTS*-from output port BA, bit 7	RTSR-to RS-232 pin 4	RTSM-to RY-2 (phone line)
TRO-from UART Transmitter Register- output port C8	TXR-to RS-232 pin 2	TXM-to modem transmitter circuitry
RRI-to UART Receiver Register- input port C8	RXR*- from RS-232 pin 3	RXM- from modem receiver circuitry
CTS*-to input port BB, bit 4	CTSR- from RS-232 pin 5	CL/AS- from SW-1 ORIG/ANS
DSR*-to input port BB, bit 5	DSRR- from RS-232 pin 6	CP/TL- from SW-2 ACP/DIR
CD-to input port D8, bit 0	logic zero	RXCAR-through 10K resistor

The items in the column headed RS-232 mode are discussed in this chapter; the items in the column headed Modem are discussed in chapter 8. The circuitry of the multiplexer is shown in figure 7.6. Curiously, the DTR signal is not switched.

It is interesting to note that the multiplexer status is one of the few things not preserved when a running BASIC program is powered down, then back up. To see this, set Stat to a numerical baud rate, and run this program in BASIC:

```
1 A=INP(187) AND 32: PRINT A: GOTO 1
```

The DIR/ACP switch has no effect on the display. Then, turn the computer OFF and back ON. Suddenly the switch will change the value on the screen.

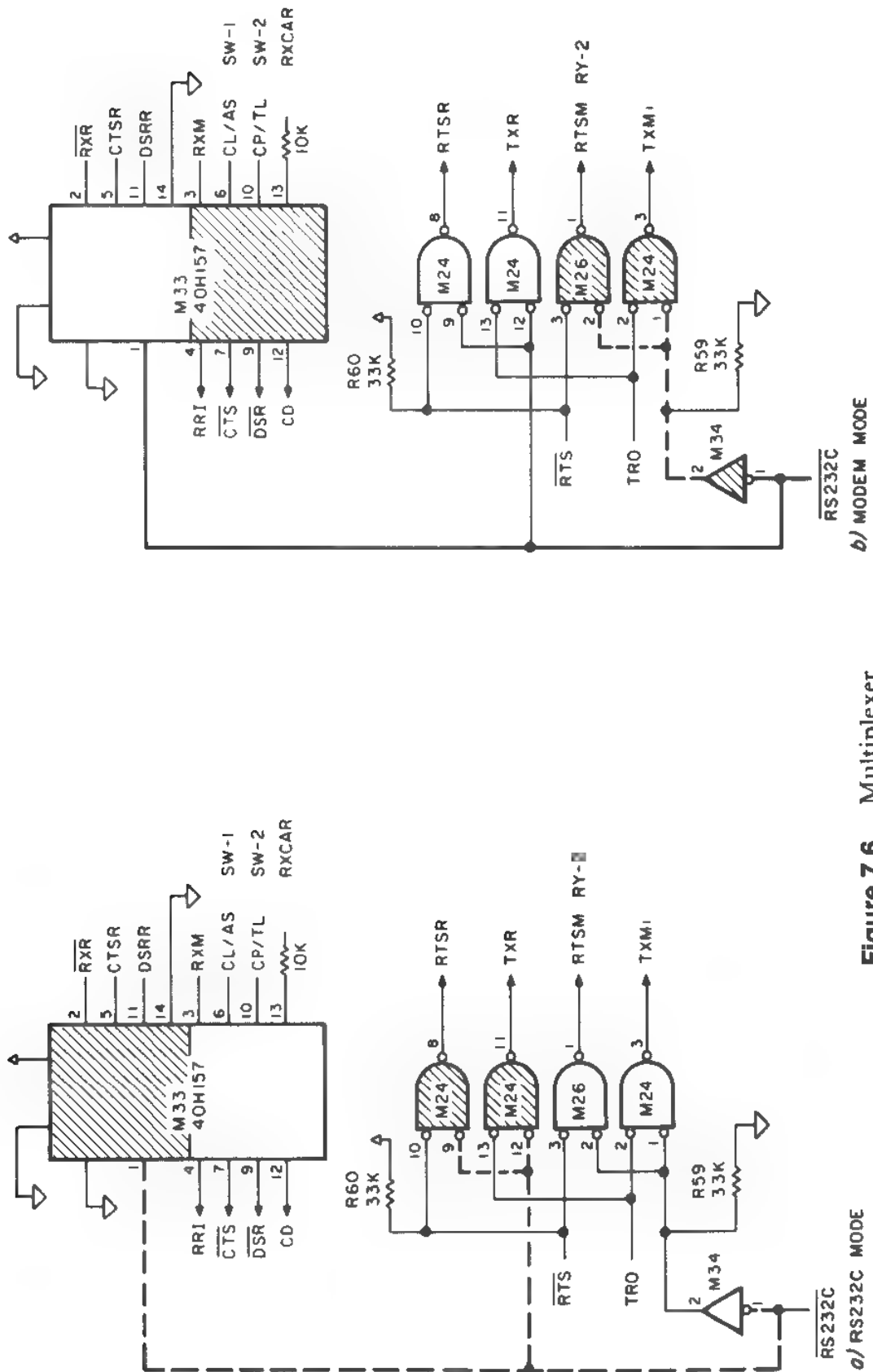


Figure 7.6. Multiplexer

## The RS-232 Standard

Decades ago the Electronic Industries Association promulgated the RS-232 standard, designed to facilitate the design of interfaces between data terminal equipment (the Model 100) and data communications equipment (modems). (The letters RS in the designation have nothing to do with Radio Shack.) The standard found widest application in the connecting of teletype-like devices with keyboard and printer (data terminals) to mainframe computers and modems (data sets). Typically, two directions of communication were intended and a variety of *handshaking* signals were designed in so that each device could inform the other whether it was able to receive and transmit data.

Since its origins, the RS-232 standard has undergone three revisions culminating in the present version, RS-232C. (Here the designations RS-232 and RS-232C will be used interchangeably.)

As we shall see, the Model 100 satisfies parts of the standard and fails to satisfy other parts. The same is true of every computer product on the market today.

### MECHANICAL REQUIREMENTS

The most visible aspect of the RS-232 standard is the specified connector, a twenty-five-pin plug and jack in the DB style. The terminal is supposed to accept the male connector, so the Model 100 violates that requirement.

Most of the twenty-five pins have defined functions, but only eight still have common use. Plugs, jacks, and cables are commonly available; many RS-232 cables have only a few of the pins wired from one end to the other. The plug is Radio Shack catalog number 276-1547 or 276-1559; this will plug into the Model 100. Unfortunately the hood, 276-1549, may not be used as it will not fit into the case of the Model 100. The jack is catalog number 276-1548 or 276-1565. A complete cable suitable for Model 100 use can be purchased ready-made (26-1408) or assembled by hand (276-1551, 276-1559, and 276-1565).

### ELECTRICAL REQUIREMENTS

Each signal in the twenty-five-pin connector has a specified originator and recipient, either the data set or the data terminal. As to each

signal, the originator is obligated to use -5 volts or less to mean a logic 1 (or *denial* of a handshake signal) and +5 volts or more to mean a logic 0 (or *assertion* of a handshake signal). The originator must meet this voltage requirement even under a 3000-ohm load. The originator is forbidden to use voltages between +5V and -5V for any purpose other than during the brief instant of transition from positive to negative, or vice versa.

It is a fact of the Model 100 design that its 0 and 1 states are a mere five and minus five volts, respectively. Under a 3000-ohm load, the voltages drops to well under five volts. As a result at the other end of the cable, the signals can easily be in violation of the RS-232 electrical requirements. This is not usually a problem since devices at the other end are likely to be able to receive the signals with as little as a one-volt swing about zero.

In the absence of a connection to the Model 100 RS-232 connector, the hardware reads all incoming signals as negative voltage, or logic "1's." When an incoming signal on any of those pins becomes more positive than about 1.5 volts, the value presented to the CPU changes to a logic 0. (This betters the threshold promised by the Radio Shack specification, which is 3 volts.) Thus, assuming a reasonably short cable, one Model 100 will have no trouble detecting the plus five volt and minus five volt signals from another Model 100.

The RS-232 standard requires that the inputs withstand as much as positive or negative 25 volts, but according to the Radio Shack Service Manual, the RS-232 inputs are designed to withstand only eighteen volts. The inputs are also required to give off no more than plus or minus two volts in open circuit; in the Model 100 they give off minus five volts.

A further RS-232 requirement is that pin 7, called AB, should be the signal ground used by each device as the zero voltage reference for the signals originating at the other end. Each device is to use that pin as the ground reference for the drivers sending voltages to the other end.

As a separate matter, pin 1, with RS-232 designation AA, is to be used as a protective (earth) ground to minimize the possibility that a person touching the chassis of the two devices would be injured by electric shock due to a voltage difference between the two. The specification requires that it be possible for the user to either tie pins 1 and 7 together or separate them within the computer.

Unfortunately, in the Model 100 the two pins are tied together permanently. This reduces noise immunity for both directions of serial data.

### DATA FLOW

According to the standard, the data set sends out data on pin 3, and the data terminal sends out data on pin 2. The signals are named from the terminal's point of view, with the pin 2 signal called BA (transmitted data) and the pin 3 signal called BB (received data). The Model 100 acts like a data terminal, talking on pin 2 and listening on pin 3. (If it is to talk with another Model 100, a so-called *null modem* is required, which switches lines 2 and 3 between two connectors. You can easily make one yourself.)

### HANDSHAKING SIGNALS

The terminal indicates its powered-up status and requests that the modem access the phone line by asserting the DTR (data terminal ready, RS-232 designation CD) signal provided to the modem at pin 20.

If the data set were a modem, it would let the data terminal know when it is powered up and connected with the transmission line by activating what is called the DSR (data set ready) line, pin 6, which has the RS-232 designation CD.

Before the terminal would send a character to the modem (to be transmitted down the phone line) it would ask permission of the modem by activating RTS (request to send), pin 4, which has RS-232 designation CA. Assuming the modem is able to send another character, it grants permission by activating CTS (clear to send, pin 5, RS-232 designation CB), which is received by the terminal.

In the Model 100, as we shall see below, hardware provision has been made for the signals named above. The TELCOM software and the ROM routines ignore DSR and CTS, which come in from outside, and assert RTS and DSR to the outside device. (When power is applied to the Model 100, RTS and DSR are not asserted. Then when the TELCOM program is run and the TERM mode is selected (function key f4), RTS and DSR are both brought to a positive voltage (asserted).

### HOW THE RS-232 INTERFACE WORKS

Handshake signals and serial data enter and leave the Model 100 through the RS-232C connector, as shown in table 7.6.

**Table 7.6.** RS232 signals handled by the Model 100

RS232 Pin, Des	Mod 100 Symbol	Source	Port Location
1-AA	(protective ground)		
2-BA	TX	Inside	Out C8
3-BB	RX	Outside	In C8
4-CA	RTS	Inside	Out BA, bit 7
5-CB	CTS	Outside	In BB, bit 4
6-CC	DSR	Outside	In BB, bit 5
7-AB	(signal ground)		
20-CD	DTR	Inside	Out BA, bit 6

These signals, according to the RS-232 convention, are negative for a data one or non-asserted value, and positive for a data zero or asserted value. However, the internal circuitry of the Model 100, like all computers, uses +5 volts for a logic one and ground or 0 volts for a logic zero level. The conversions, three incoming and three outgoing, are performed by integrated circuit M35. The RS-232 interface circuitry is shown in figure 7.7.

The incoming handshake signals, clear-to-send and data-set-ready, are then made available to the CPU at bits 4 and 5 of input port BB. A negative voltage at the input pin appears to the CPU as a zero at the input port, while a positive voltage comes through as a one.

The outgoing handshake signals, data-set-ready and request-to-send, are controlled by the CPU through bits 6 and 7, respectively, of output port BA. In each case, 0 and 1 sent out by the CPU result in positive and negative voltages, respectively.

### RECEIVING RS-232 DATA

Serial data enters the Model 100 through RS-232 pin 3 and, after level shifting and signal switching, is fed to the UART.

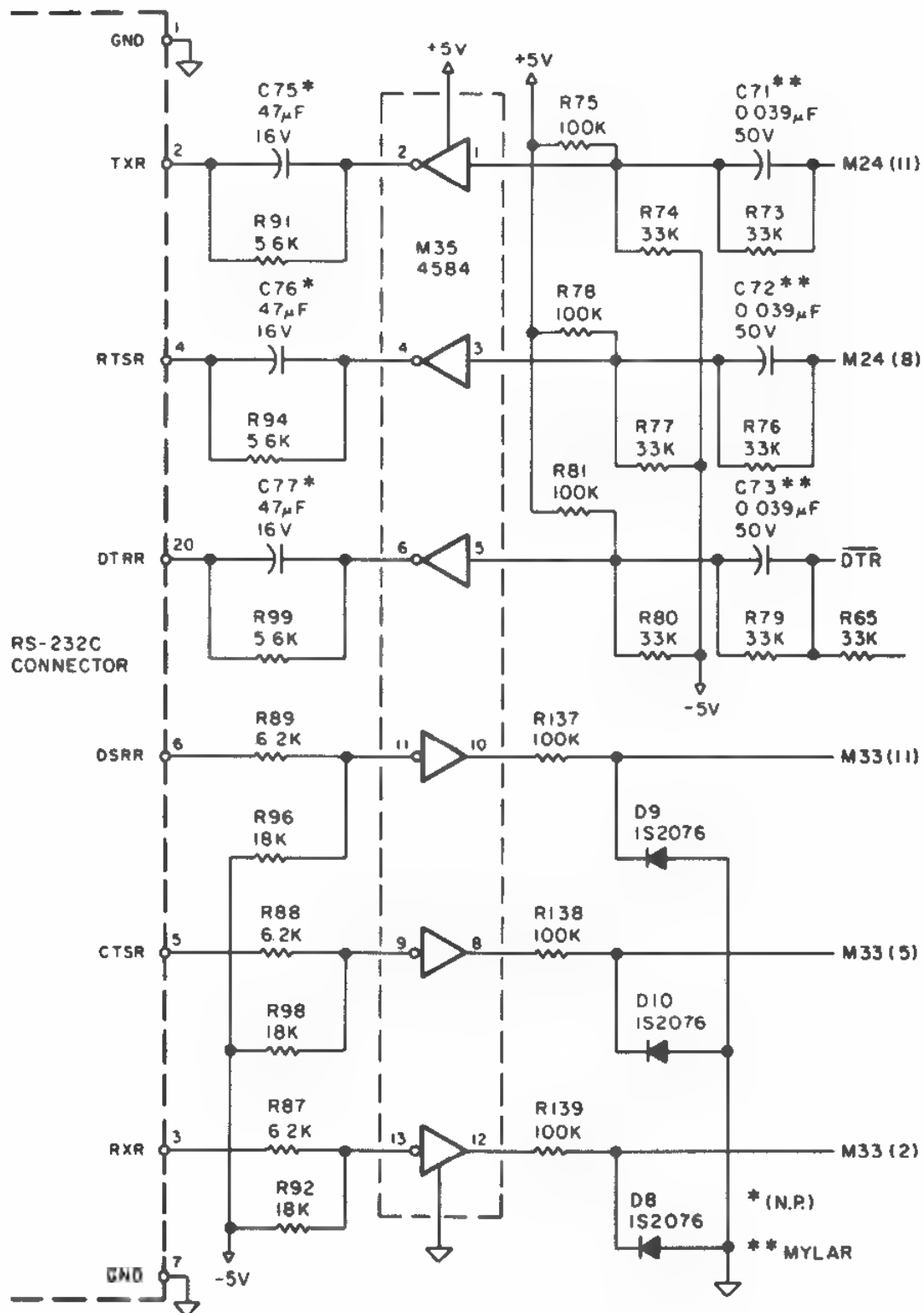


Figure 7.7. RS-232 interface



## **TRANSMITTING RS-232 DATA**

The serial output from the UART goes through switching circuitry to RS-232 level shifter M35, and from there to RS-232 pin 2.

## **PUBLISHED ROM SUBROUTINES**

The UART baud rate can be set by the routine BAUDST, called at 6E75. Prior to the call, H contains a numerical value in the range of 1 through 9, and the routine sets the baud rate using a ROM table at 6E94 through 6EA5. The table address used in the load is stored in FF8B. Note that the routine will not work properly if H contains undefined values, such as ASCII values for characters 1 through 9 or M. The routine extends to 6E93.

The routine INZCOM, called at 6EA6, performs the baud setting function of BAUDST and configures the UART for word length, parity, and so on. Prior to the call, H must contain the baud rate number, just as with BAUDST. In addition, L must contain "1's" and "0's" to select the UART parameters desired, from table 7.2. (Some Radio shack documentation is incorrect on this; table 7.2 should be used.) The condition of the carry flag determines whether the routine sets the multiplexer for RS-232C (if carry is set) or modem mode (if carry is reset). The routine initializes a UART data-received buffer.

It does not matter what bit values are chosen for bits 5 through 7 of the L register; the hardware ignores them and the ROM routine trims them off anyway.

The routine SETSER, called at 17E6, sets the baud rate and UART word length parameters based on an ASCII text string similar to the one which follows the device designator COM: or MDM:. The routine also initializes a UART data-received buffer, and updates a flag located at FF42 controlling XON/XOFF. A zero at FF42 means XON/XOFF is enabled; a nonzero value means it is disabled.

Before calling the routine, HL must point to the ASCII text string. The carry flag determines whether the routine switches the multiplexer to RS-232 or modem mode, just as in the INZCOM routine. If modem mode is to be used, the D register must be loaded with the value of two prior to the call, and the text string must start not with the baud rate "M" character but with the word length digit.

The correctness of the text string is fully checked, and the alphabetic characters can be uppercase or lowercase. When the routine finishes, if the Z flag is set, something was wrong with the text string. Otherwise, you may assume the baud rate and parameters are set. There is no need for the string to end with a zero, as the routine simply reads enough bytes to get what it needs.

This routine updates the storage location STAT, at F65B through F65F, which contains in ASCII format the Stat information, baud rate (M if modem), word length, parity, stop bits, and XON/XOFF switch.

The routine CLSCOM, called at 6ECB, deactivates the RS-232 or modem circuitry. More precisely, it "un-asserts" RTS, if it is in RS-232 mode. If in modem mode, it simply hangs up the phone. In either case, it "un-asserts" DTR.

The routine RCVX, called at 6D6D, checks the UART data-received queue to see if any characters have been received since the queue was last emptied. The A register contains the number of characters in the queue, and the Z flag will be set if no received data is pending, or reset otherwise.

This routine can be used regardless of whether the Model 100 is in RS-232 mode or modem mode.

The routine RV232C, called at 6D7E, retrieves a character from the UART data-received queue. If no character was present in the queue at the time of the call, the routine does not return until a character is received or SHIFT-BREAK is pressed. (CTRL-C will not cause a return; only the precise combination SHIFT-BREAK will do so.)

The call is appropriate regardless of whether the computer is in RS-232 or modem mode. The received character is in the A register, and the conditions of the Z and C flags indicate whether errors occurred. The Z flag is set if the character was received properly, and reset if a PE, FE, or OE occurred. If the routine returned because SHIFT-BREAK was pressed, the carry flag will be set; or reset otherwise.

The routine SENDCQ, called at 6E0B, sends an XON (CTRL-Q, ASCII 11H) character to the remote device, but only if XON/XOFF was enabled at the time of UART initialization. (Recall the XON/XOFF flag at FF42 is zero if XON/XOFF is enabled and nonzero if disabled.)

The routine SENDCS, called at 6E1E, sends an XOFF (CTRL-S, ASCII 13H) character to the remote device, but only if XON/XOFF was enabled at the time of UART initialization. (Recall the XON/XOFF flag at FF42 is zero if XON/XOFF is enabled and nonzero if disabled.)

The routine SD232C, called at 6E32, sends a character to the UART to be transmitted to the remote device. The character to be sent should be present in the A register prior to the call.

It is possible for the routine to return without successfully sending the character. Suppose the UART takes so long to transmit (or the other device sends the Model 100 a CTRL-S for such a long time) that the user presses SHIFT-BREAK. The routine returns with the carry flag set.

If the UART was originally configured with XON/XOFF disabled (FF42=0) then the routine simply sends the character. If XON/XOFF is enabled, then the routine makes a point of keeping track (the flag at FF41) of which has been sent out by the Model 100 most recently— CTRL-S (flag=-1) or CTL-Q (flag=9).

Before you use the UART, it is a good practice to clear the UART receiver buffer register with an input from port C8. This is done, for example, at 6CE5.

# 8

---

## The Telephone Modem

---

The Model 100's built-in modem and autodial capabilities are among its most popular features. This chapter consists of a discussion of the modem and autodial feature. The hardware of the autodial, direct-connect modem is explained first followed by a description of the acoustically-coupled modem. ROM subroutines are included for both types of modem.

### **Data Flow Overview**

Recall from the discussion in chapter 7 that serial-to-parallel and parallel-to-serial conversions take place within the UART, and that a

multiplexer determines whether the UART is connected to RS-232 circuitry or modem circuitry. This is shown in functional block diagram form in figure 8.1.

The multiplexer, shown here in the "RS-232" position, sends incoming RS-232 data to the receiver portion of the UART which is then sent to the CPU. The multiplexer also connects the output of the UART transmitter to the RS-232 output circuitry.

If and when the multiplexer switches to the other position, the data paths change in two ways. As shown in figure 8.1a, the UART receiver gets its signal not from the RS-232 circuitry but instead from the modem receiver (which in turn gets its signal from the modem filter). The UART transmitter is connected to the modem transmitter, which in turn feeds a wave shaper, as shown in figure 8.1b.

Assuming the multiplexer is in "modem" mode, then the DIR/ACP and ORIG/ANS switches also have an effect on the data paths.

The DIR/ACP switch determines whether the modem filter gets its signal from the phone line transformer OT1 or from the electret-condenser microphone in the earpiece cup of the acoustic coupler. In addition, it connects the output of the wave shaper either to OT1 or to a tiny speaker in the mouthpiece of the acoustic coupler.

Finally, the ORIG/ANS switch changes the internal function of the modem filter, the wave shaper, and the transmitter and receiver portions of the modem integrated circuit.

The various circuits shown in the functional block diagram will be explained in detail, but first it will be necessary to explain a bit about the Bell 103 standard, and about telephones generally.

## **The Bell 103 Standard**

All 300 baud data transmission in North America is performed according to Bell Standard 103, which spells out the manner in which computers are to send 1's and 0's to each other.

The technique used is frequency-shift keying (FSK), which means that each computer produces an audio tone, and shifts in the frequency of the tone indicate whether a "1" or "0" is being sent. The Bell 103 standard calls for each computer to transmit a "1" most of the time, dropping the frequency by 200 Hertz whenever a "0" is to be sent.

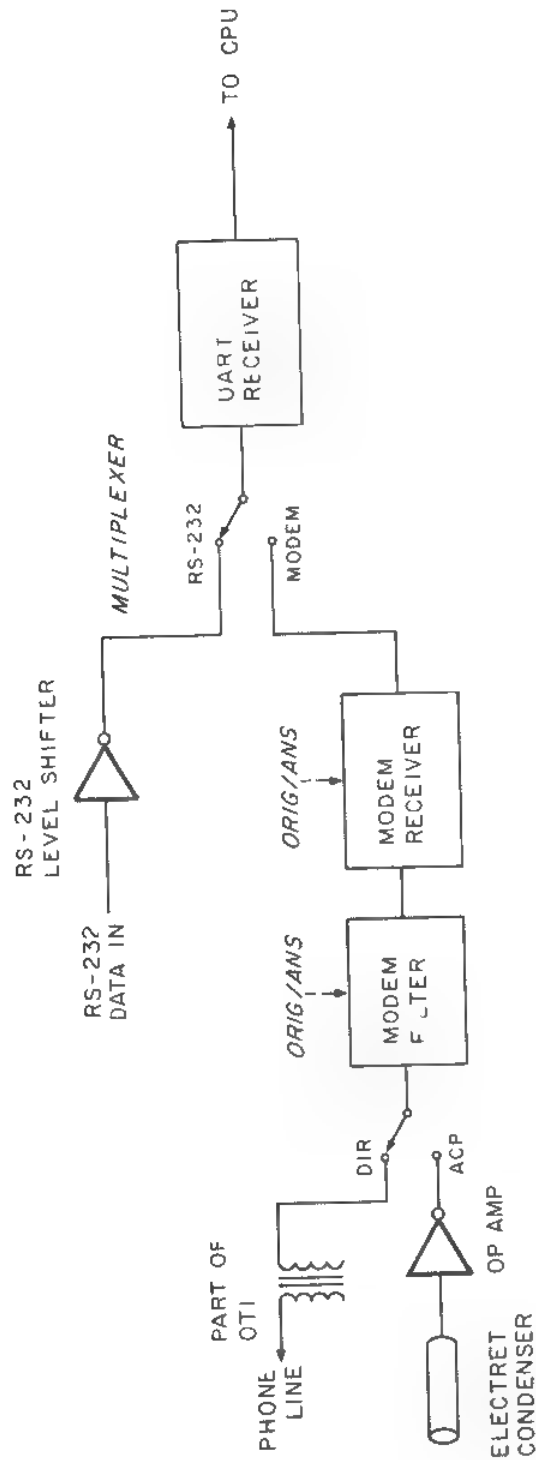
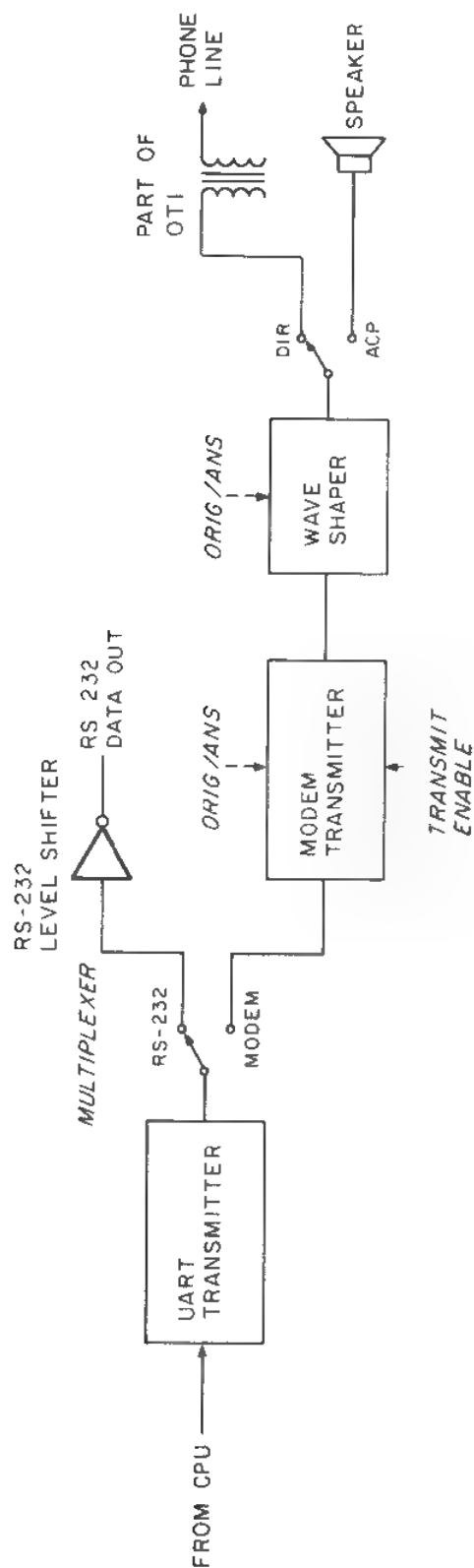


Figure 8.1a. Receiver data path



**Figure 8.1b.** Data path switching functional block diagram

The "1" state is called "marking," and the "0" is called a "space". This terminology originated with the first teletype machines. Historically the usual circuit condition between two teletypes was a current flow of 20 milliamperes, called a *mark*. Brief interruptions in the current were called *spaces*. An analogy can be drawn between frequency-shift keying and the dialing of a phone.

To this day, most telex machines use this so-called *current loop* means of data communication which allows them to be interfaced with any RS-232 device such as the Model 100, by using optoisolators.

### THE AUDIO FREQUENCIES

In order for two-way data communication to take place, each computer must be prepared to listen carefully to the audio tone transmitted from the other device, to determine whether a "1" or "0" is being sent. At the same time the computer must generate audio tones to send 1's and 0's to the other device.

Problems arise when both computers use the same tone because each computer will hear itself as well as the other computer. The solution is to assign different frequencies to the two devices.

When the Bell 103 standard was written, most modem communications was quite lopsided. One device was a large computer, which answered calls, while at the other was a small device, usually a terminal, which originated calls. The originating device was assigned a mark frequency of 1270 Hertz, while the answering device was assigned 2225 Hertz. The established protocol was that the answering device, upon answering the phone, would emit the 2225 Hertz tone. Upon hearing it, the originating device would start producing the 1270 Hertz tone.

The two tones, known as *carrier* tones, were expected to be present (at either the stated frequency or 200 Hertz down) during the duration of the phone call. If either computer's tone disappeared for even an instant, the other device would assume it had been hung up on, and would disconnect itself. This is the reason that the call waiting feature causes so much trouble for modems.

Currently in many situations (such as communications between two Model 100's) the selection of one device as the *originate* device and the other as the *answer* device is purely arbitrary.



The other major standard, used in Europe, is the CCITT standard. It is similar in most respects to the Bell 103, except for the particular frequencies used to represent 1's and 0's.

## **How Telephones Work**

### **THE PHONE WIRES**

Although standard modular telephone jacks contain four wires—red, green, yellow and black, most telephones use only two of them, the red and green wires. The red wire is sometimes called the *ring* signal, and the green wire is sometimes referred to as the *tip* signal. The terms ring and tip have nothing to do with the ringing of the phone bell; they come from the physical description of the barrel (ring) and end (tip) of the two-conductor phone plugs traditionally used by switchboard operators in connecting calls.

### **ELECTRICAL CONSIDERATIONS**

If no phone is plugged into the jack, or if a phone is plugged in and *on-hook* or hang-up, the potential on the line will be about forty volts. An on-hook phone represents a very high resistance across the two wires, so that virtually no current flows.

### **PLACING TELEPHONE CALLS**

When the phone is taken *off-hook*, that is, when it is picked up to place or answer a call, the phone presents a low resistance across the two wires. The central office detects this, and presents a dial tone or connects the incoming call, whichever is appropriate. The low resistance is typically 600 ohms. With such a load on the line, the voltage from the central office drops substantially to perhaps ten volts.

Once a dial tone is audible, the next step is generally dialing a number. One of two methods may be used, depending on the nature of the dial tone circuitry provided. With all dial tones, one may use rotary dial pulses; with some dial tones, DTMF (dual tone multifrequency, e.g. Touch-Tone) may also be used.

Rotary dial pulses are sent to the central office by repeatedly removing from the phone line the low resistance path that was present

when the phone was taken off-hook. In a traditional rotary-dial phone, this is accomplished using a simple mechanism of springs, gears and switches. With many modern pulse-type, pushbutton phones, solid-state circuitry mimics the dial mechanism.

There is nothing mysterious about the action of the telephone dial. It merely hangs up the phone (places it on-hook) one or more times for only a fraction of a second. You can do this manually with any phone by simply tapping the hang-up button. (The on-hook and off-hook times should last about 65\* and 35\* milliseconds, respectively.) After a pause of about 300\* milliseconds, the next digit is dialed.

### **ANSWERING TELEPHONE CALLS**

Now that the number has been dialed, let's examine what takes place when the phone rings. The central office causes a phone to ring by sending AC (alternating current) at perhaps forty or fifty volts to the phone jack. The telephone uses a coupling capacitor to allow the AC to ring the bell.

### **RINGER EQUIVALENCE**

The amount of energy absorbed by a device connected to the phone line when the ringing voltage is present is reflected in the ringer equivalence number (REN) of the device. An REN of 1 means the device takes as much power as a traditional Western Electric phone. The Model 100 has a REN of zero, because when relay RY2 is open the computer has no current path capable of absorbing an appreciable amount of the ringing energy.

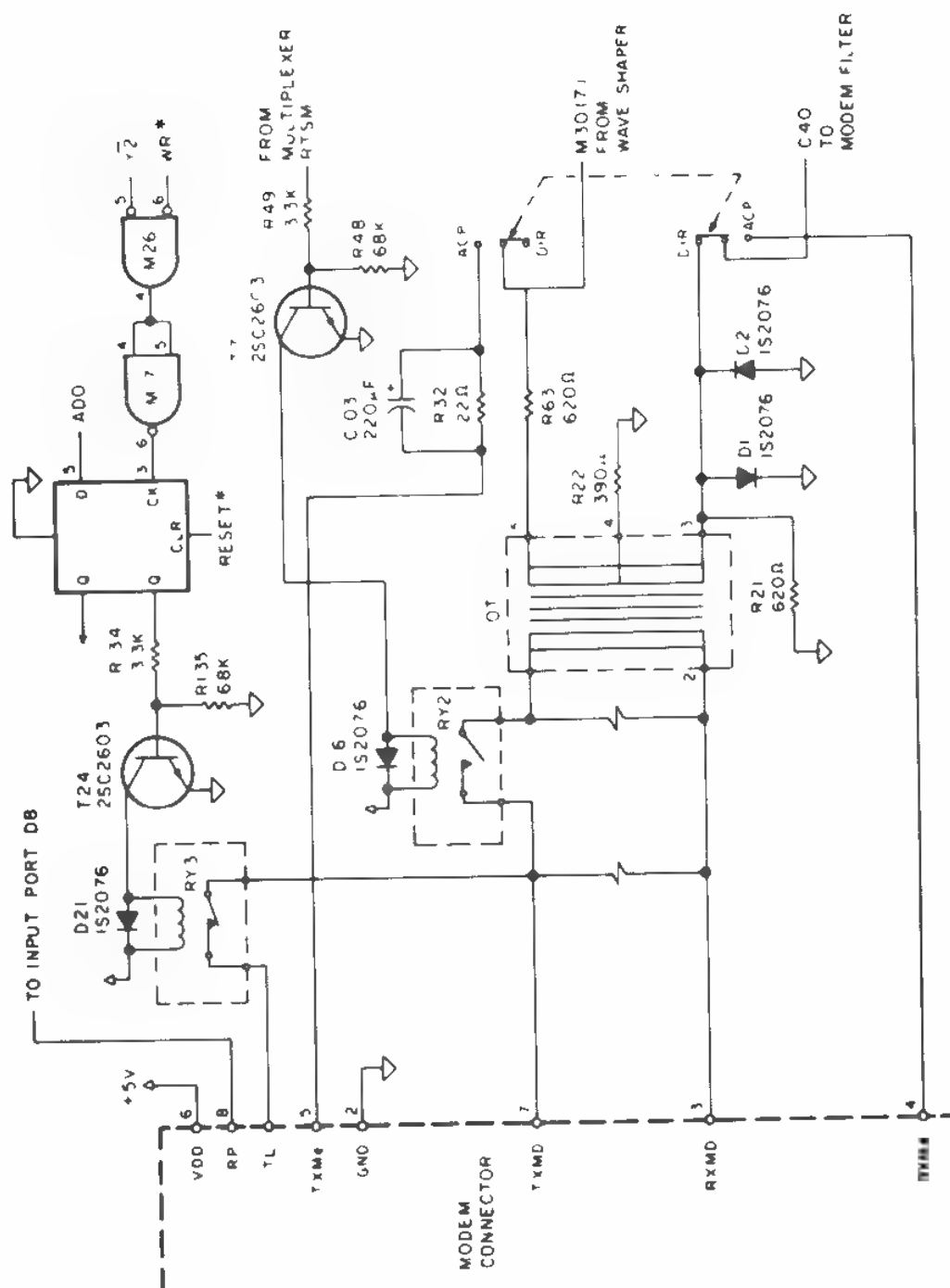
Simply taking the phone off-hook signals to the central office to stop sending the ringing voltage and to connect the calling party.

### **THE DIRECT-CONNECT MODEM AND THE TRANSFORMER OT1**

The Model 100 uses the red and green wires for direct-connect modem operation. The red wire enters through pin 7, as shown in figure 8.2.

---

\* For 20 pulse-per-second dialing, all these times are divided by two.



**Figure 8.2.** Direct connect interface

Two relays control the direct telephone connections of the Model 100. Though it is normally open, relay RY2, when energized, connects the primary side of isolation transformer OT1 to the tip and ring signals. Relay RY3, which is normally closed, makes the connection between the telephone instrument and the telephone line. Relay RY2 accomplishes the resistance transition from infinity down to six hundred ohms to pick up the phone, and to dial phone numbers. Relay RY2 could be termed the *hookswitch* relay.

Recall that modem cable 26-1410 has two modular plugs, with beige and silver cords. The beige cord plugs into a modular jack, bringing the tip and ring signals to pins 3 and 7 as mentioned above. The ring signal at pin 3 goes directly to the phone instrument (if connected) through the red conductor of the silver modular cord. The tip signal from the beige cord enters the Model 100 at pin 7 and is usually fed through relay RY3 to pin 1. (These signals are summarized in table 8.1.) From there the tip signal passes through the silver cord's green conductor to the instrument.

**Table 8.1.** Phone jack pin designations.

Pin	Designation	Function
1	TL	Green to phone instrument
2	GND	Ground reference
3	RXMD	Red from phone line ("ring" signal) also goes to phone instrument
4	RXMe	From coupler earpiece electret-condenser microphone
5	TXMe	To coupler mouthpiece
6	VDD	+5V for coupler amplifier
7	TXMD	Green from phone line ("tip" signal)
8	RP	Ring pulse signal (see text)

When direct-connect modem data transmission is desired, it is necessary to energize both relays. (This is accomplished by the TELCOM software.) This connects the phone line to the modem circuitry and disconnects the telephone instrument to avoid interference if the phone is picked up. It is impossible, of course, for the Model 100 to protect against interference caused by picking up an extension phone on the same line, or from problems caused by loss of the audio carrier signal such as a call-waiting beep.

When TELCOM is used as an automatic dialer for voice conversations, relays RY2 and RY3 are both energized. Relay RY2 is left open for about a second to insure that any previous call is disconnected and is then turned on again. After allowing a couple of seconds for the dial tone to arrive, RY2 is repeatedly switched on and off to simulate a rotary telephone dial. Then both relays are de-energized, leaving the phone instrument connected to either ringing or a busy signal.

The dialing process is simple. To dial, say, a "4," relay RY2 is switched off and on four times.

### **RING PULSE**

Provision has been made for the Model 100 to be expanded into an autoanswer device. This prospect is discussed further in chapter 17.

### **FCC CERTIFICATION**

On the bottom panel of the Model 100 are labels describing two kinds of FCC certification. The first, which bears FCC identification number AWQ9SB26-3802, indicates that the computer has been tested and found to be sufficiently shielded. This means that it does not radiate in excess of levels of radio frequency (RF) energy set forth in part 15 of the FCC rules. The most noticeable part of the shielding is a foil panel, resting between the main printed circuit board and the black plastic at the bottom of the case.

The other FCC certification, number AWQ9SB-70372-DT-R, pertains to the physical and electrical qualities of the circuitry shown in figure 8.2. The standard used is referred to in part 68 of the FCC rules, which requires that the computer must not interfere with the ability of other phone customers to place their calls and must not generate any voltages that might injure telephone workers.

Neither of these FCC certifications establishes that the computer does a proper job of dialing the phone nor does it even indicate that the computer will function when it is turned on. (An empty box would also satisfy both FCC requirements and would in fact be easier to get certified.)

### **MODEM DATA FLOW**

The incoming modem signal passes through a filter composed of six operational amplifiers, shown in figure 8.3. The filters remove almost everything except the energy in the neighborhood of the frequency of the incoming carrier. In the originate mode, this is 2025-2225 Hertz; in answer mode this is 1070-1270 Hertz. (Transistors T2, T3, and T5 affect the frequency change in the filter.) The resulting signal, designated RXCAR, varies between 0 and 5 volts and wiggles up and down at the same frequency as the received carrier. It goes to the modem chip and appears as a "1" (about half of the time) at bit 0 of input port D8. The CPU can check to see if the carrier is being received by noting whether RXCAR keeps changing (carrier present) or remains constant always 0 or always 1 (carrier absent).

### **THE MODEM RECEIVER**

The output of the modem filter goes to the modem integrated circuit, shown in figure 8.4. It accepts the signal from the modem filter, which ranges from 0 to 5 volts, and which varies in frequency. Based on the position of the ORIG/ANS switch, it interprets the signal to yield serial digital data. For example, in the originate mode, if the RXCAR signal is 2225 Hertz, the modem chip will send a logic "1" to the UART on the RXMi line.

The internal structure of the modem chip is shown in figure 8.5. The *type* input, at pin 14, can configure the chip for CCITT frequencies. For conversion to CCITT, however, the Model 100 would also require changes in the modem filter and wave shaper.

The modem chip includes a pin, TTL<sub>D</sub>, which reduces power consumption in the chip when, as in the Model 100, it is connected only to CMOS components.

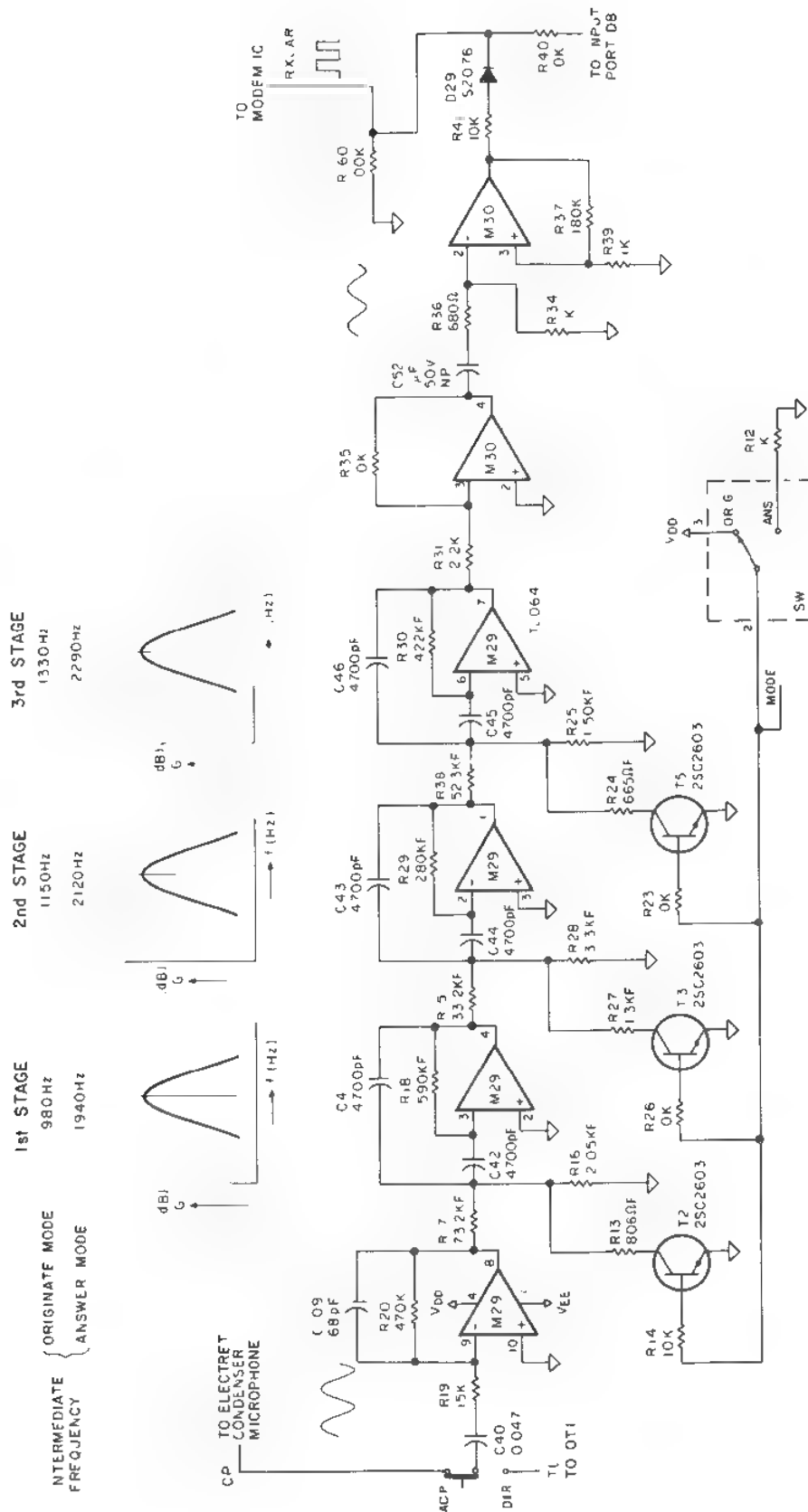
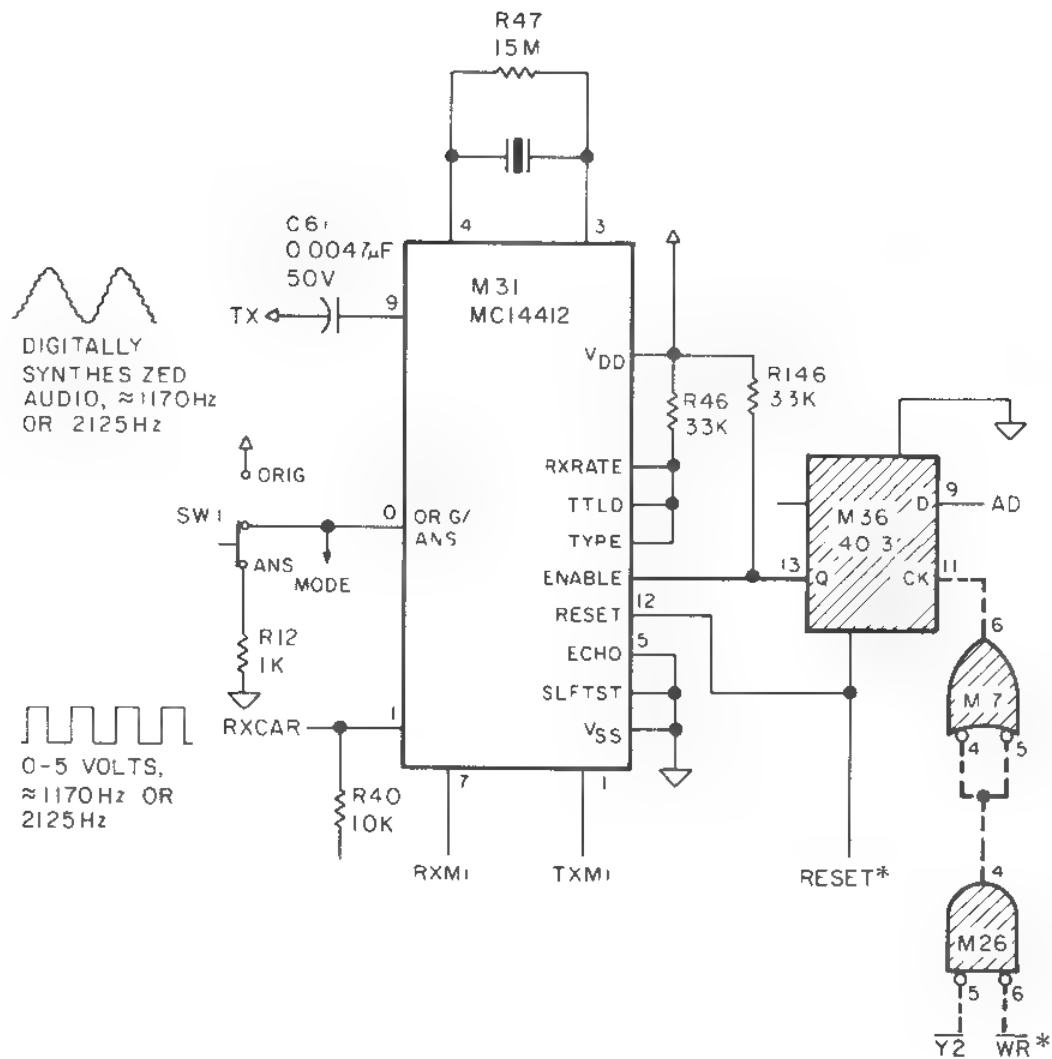


Figure 8.3. Modem incoming data filter



**Figure 8.4. Modem IC connections**



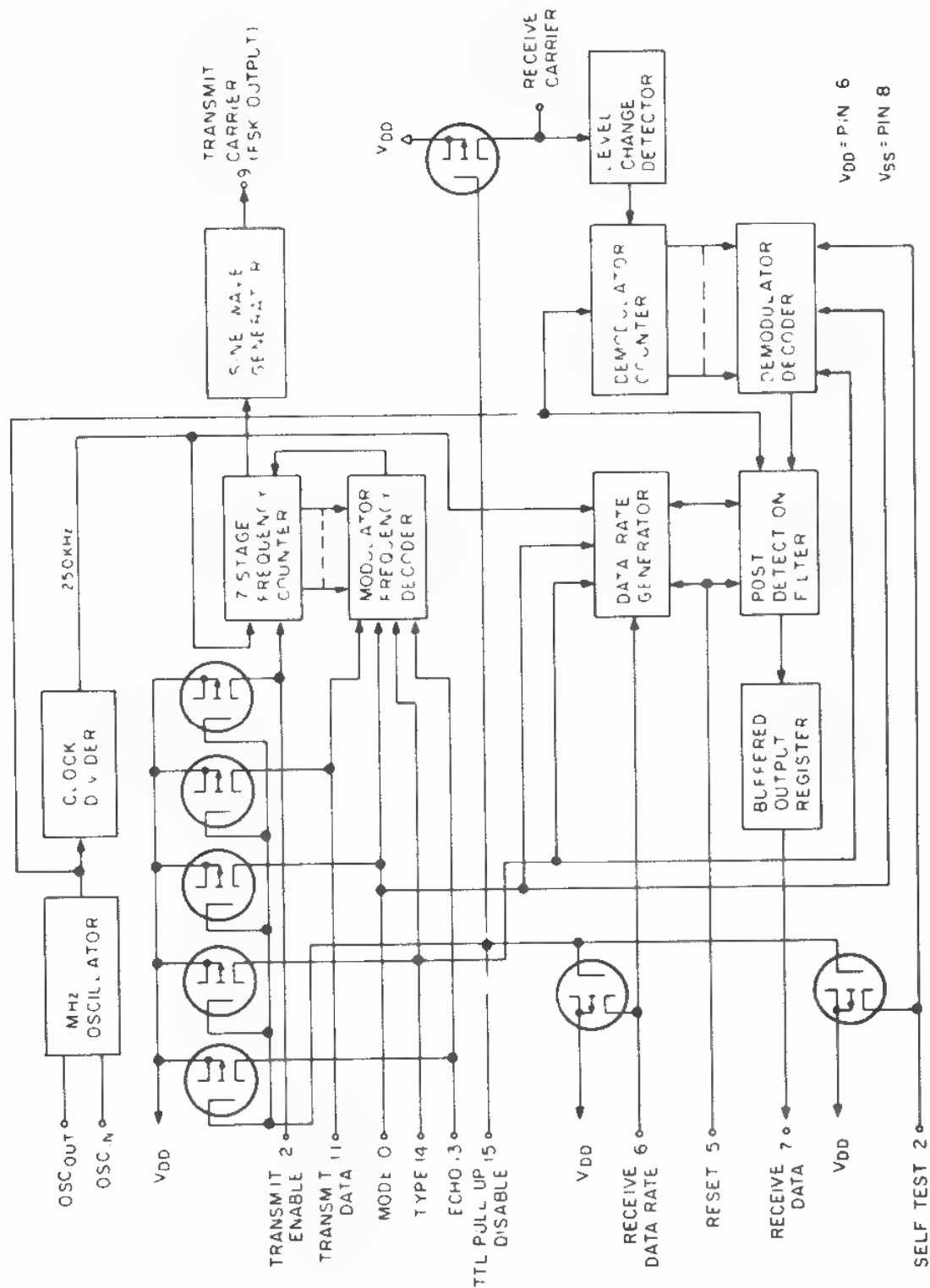


Figure 8.5. Modem IC functional block diagram

The resulting digital signal RXMi goes to the UART, where it is treated just the same as in the RS-232 mode discussed in the previous chapter.

Before modem output can be performed, the CPU must enable carrier output by means of a "1" at bit 1 of output port A8. (For most modems in the originate mode, the usual practice is to send out the originate carrier signal only after detecting the incoming carrier signal.)

### **OUTGOING DATA PATH**

Outgoing modem data is sent by the CPU just as it was in the RS-232 mode. From the UART the serial digital signal TXMi goes to the modem chip, as shown in figure 8.4. The output, a synthesized audio signal TX, is fed to the wave shaper circuit shown in figure 8.6. There the audio volume level is set by potentiometer VR2 and is amplified and sent down the telephone line through OT1 or the coupler mouthpiece, depending on the position of the DIR/ACP switch.

### **ACOUSTICALLY-COUPLED MODEM**

For acoustic-coupler operation, acoustic coupler 26-3805 is connected to the Model 100 instead of the direct-connect modem cable. When the DIR/ACP switch SW-2 is moved to the ACP position, three things happen.

First, assuming the Model 100 is in the RS-232C mode, bit 5 of input port BB will be found to be 1 rather than 0. It is this bit that allows the CPU to know that the DIR/ACP switch has been moved to the ACP position, although the Model 100 ROM routines never put this information to use.

Second, the computer "talks" to the coupler rather than to the direct-connect cable. The modem audio output signal TX is removed from the direct-connect matching transformer OT1 and goes instead to pin 5 of the phone jack, and from there to the acoustically-coupled speaker that clamps to the mouthpiece of the telephone handset. The modem cable connects to the mouthpiece cup by means of a 3/32" plug (similar to Radio Shack cat. no. 274-289) and jack. This is shown in figure 8.7.

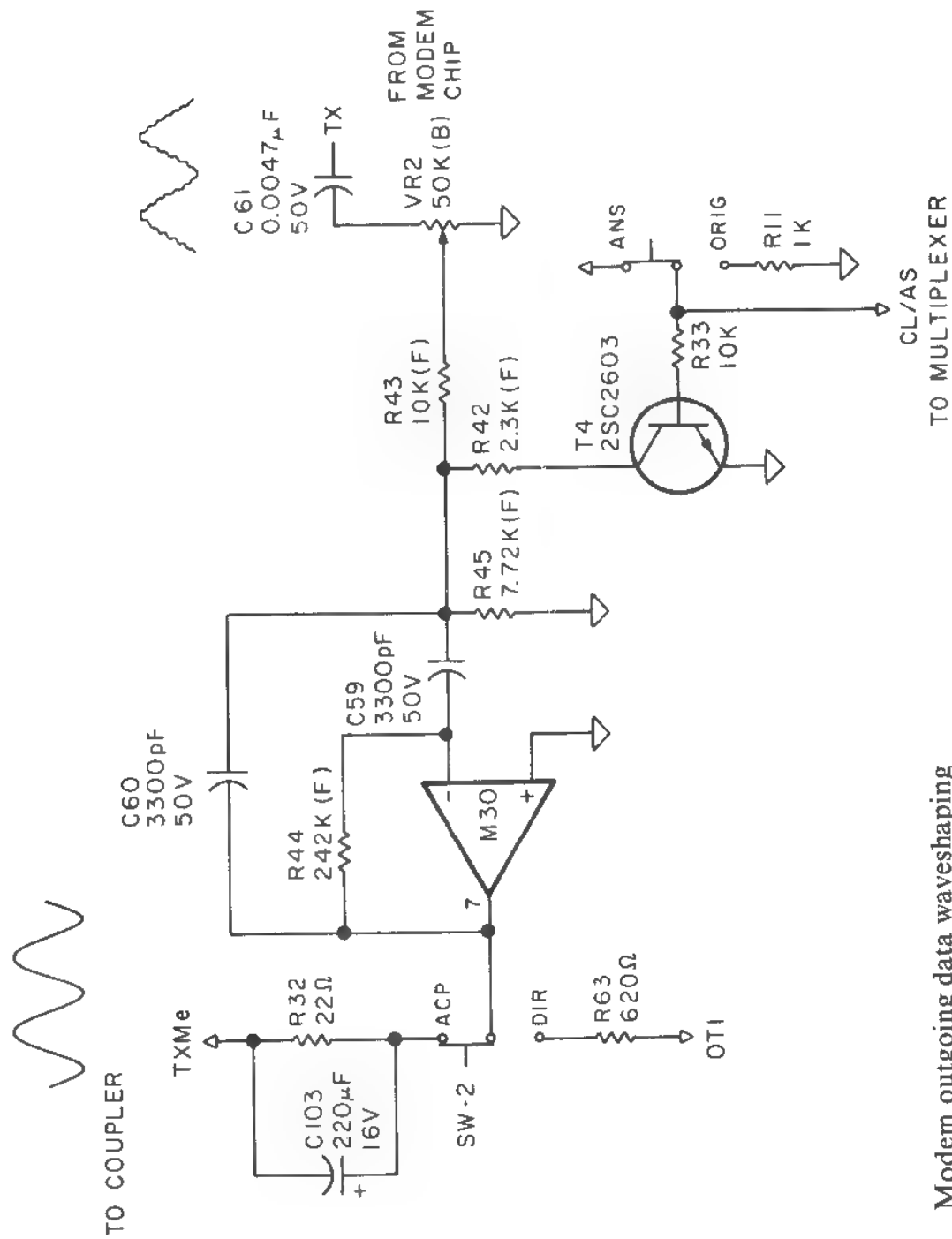


Figure 8.6. Modem outgoing data waveshaping

Finally, the computer "listens" to the coupler rather than to the direct-connect cable. The modem audio filter receives its input from the earpiece of the handset via an acoustically-coupled electret-condenser microphone, shown in figure 8.7, and not from OT1. The microphone signal is amplified by an operational amplifier (located in the earpiece cup) which draws upon the 5 Volt power available at phone jack pin 6. Capacitor C3 filters ripples from the 5 volt supply, while C2 removes high frequency pickup. Resistor R9 provides DC power to the microphone itself, while C4 capacitively couples the audio signal to the op amp.

The amplifier audio output is provided for the Model 100 at phone jack pin 4. Both the speaker and microphone are grounded at pin 2.

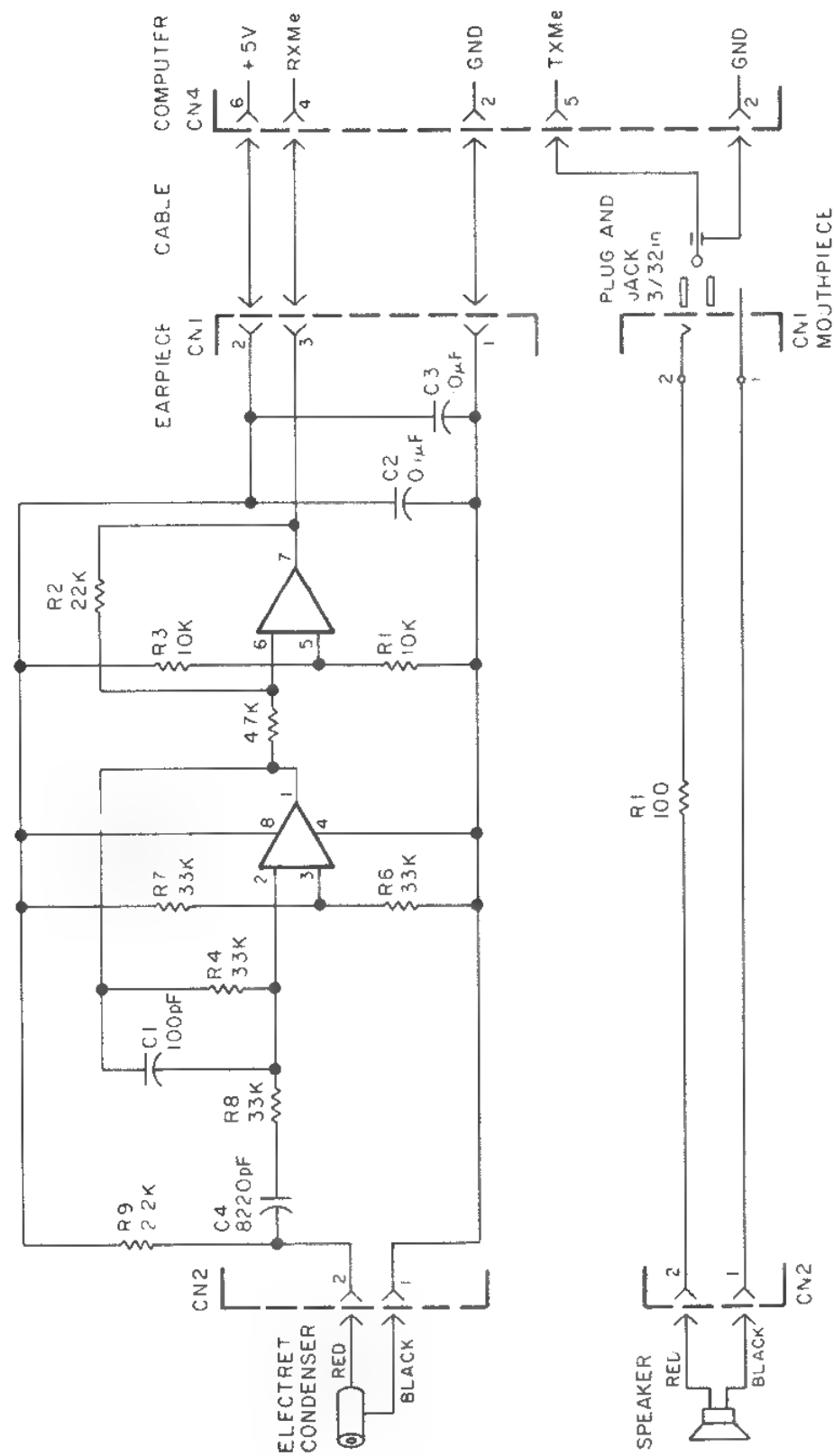
The acoustic coupler plug uses pins 2, 4, 5 and 6. By comparison, the direct-connect modem cable uses pins 1, 3 and 7. In each case, the cable requires an uncommon 8-pin DIN plug which differs from a standard 5-pin DIN plug such as Radio Shack carries (*see* catalog number 274-003).

### **USE OF THE COUPLER**

When the acoustic coupler is plugged into the Model 100, the line-control relays RY2 and RY3 are not connected to anything. Thus the autodialing features of TELCOM cannot be put to use, and dialing a computer access number must be performed manually. It would have been more efficient if TELCOM had been written to sense the position of the DIR / ACP switch, so that when ACP was selected, the computer would skip the autodialing process and go directly to the login sequence. Instead in the ACP mode, TELCOM still goes through the futile routine of dialing the phone number.

### **DIALING PROCEDURES WITH THE COUPLER**

When using the coupler, one dials the phone number and then connects the acoustic cups. When the carrier tone is audible, one choice is to push the "Term" key of TELCOM, which establishes the connection of the Model 100's modem circuitry to the cups. However, it would be nice to take advantage of TELCOM's ability to send the login



**Figure 8.7.** Acoustically-coupled Modem

sequence. This can be accomplished by using TELCOM's "Call" feature even though the dialer does not function. When TELCOM detects the carrier signal, it goes to the login sequence contained in the angle-brackets "<>" just as it would if the direct-connect cable had been attached.

## I/O PORTS

Most of the modem functions are in ports shared with other functions. One port, A8, is used exclusively for modem functions. See table 8.1. RAM location FAAE contains the present contents of the port, to facilitate changing one bit without affecting the other. Other modem I/O functions are listed in table 8.2.

**Table 8.1.** Telephone relay/modem control (output port A8; contents at FAAE)

Bit	Function
0	Telephone instrument relay (1=disconnect)
1	modem transmit (1=enable)
2-7	not used

**Table 8.2.** Modem I/O port functions

Port	Bit	Function
out BA	7	phone line off-hook
in BB	4	1=ANS, 0=ORIG
in BB	5	1=ACP, 0=DIR
in D8	0	Carrier Detect
in D8	5	Ring Pulse

## ROM SUBROUTINES

Four subroutine addresses have been published for modem operations. Two, CARDET and DIAL, are of substantial value, while the other two are simple implementations of the I/O port functions discussed earlier.

The routine CARDET, called at 6EEF, returns with the Z flag set and A=00 if a carrier is detected. It returns with Z reset and A=FF if no

carrier is detected. CARDET, which lies in ROM at 6ED6-6F30, and uses some rather tricky techniques. At 6EF2, for example, the value 6F2C is pushed to the stack, and if the search for carrier is unsuccessful, a RET instruction results in a jump to that address, which is halfway through another opcode. (Normally, every PUSH has an associated POP that is always executed.)

6EE5-6EED contains code which toggles the beeper during the carrier search if SOUND is ON (i.e. (FF44) is zero).

Other published routines are as follows:

DISC Called at 52BB, disables transmission of carrier, reconnects the phone instrument, and puts the phone line back on-hook.

CONN Called at 52D0, takes the phone line off-hook, disconnects the phone instrument, and enables carrier transmission.

DIAL Called at 532D, dials a phone number and follows a login sequence, just as does the "Call" button in TELCOM. Before the call, HL must point to the phone number sequence.

If the sequence has a CTRL-Z, CR or LF before an angle bracket (" $<$ ") the routine finishes by connecting the phone instrument; thus DIAL may be used as an autodialer.

Upon return from the routine, if the carry flag is set, the routine was unsuccessful, probably because the SHIFT-BREAK key was pushed.

The dialing rate, 10 or 20 pps, is a function of the "pps" flag at F62B.

## 9

---

### Piezoelectric Beeper

---

Located directly under the TRS-80 top panel logo is a piezoelectric beeper. It provides an audio monitor of cassette data input and TELCOM dialing progress and performs the BASIC commands SOUND and BEEP.

#### **How Piezo Beepers Work**

Since the 19th century it has been known that pressure applied to certain crystals generates electricity. This piezoelectric effect is named after the Greek word “piezein” meaning to press. Years ago the effect was used in crystal microphones and crystal phonograph cartridges; one common present-day consumer application is the flintless butane lighter, in which mechanical energy from the user’s thumb is converted to a voltage high enough to create a spark to light the vaporized fuel.



A lesser-known aspect of the piezoelectric effect is the fact that application of electrical potential to such a crystal causes physical deformation, such as expansion, contraction, or twisting, depending on the shape of the crystal and the location at which the potential is applied.

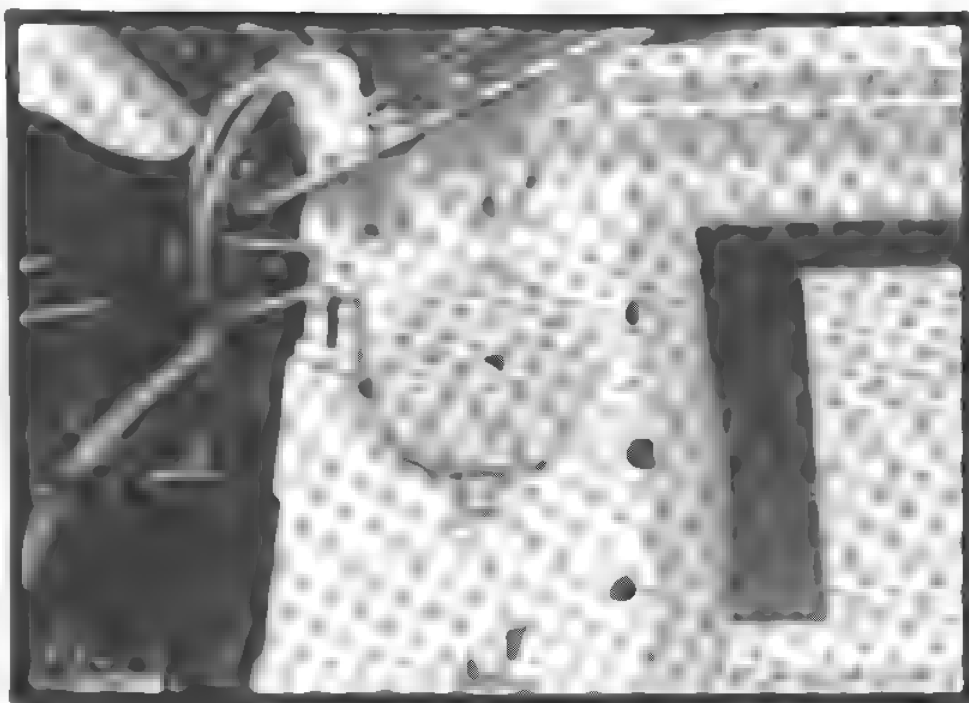
The most common consumer applications of the electrical-to-mechanical principle are the little earphones provided with inexpensive transistor radios and the high-pitched chirpers used in smoke detectors and one-piece telephones.

It is this latter aspect of the piezoelectric effect that is used in piezo beepers. When stimulated by a varying voltage, a quartz crystal deforms, moving a metal disc to which it is attached. For example, a square wave electrical signal applied to the disc produces something approaching a square wave audio signal in the air, because of the movement of the disc.

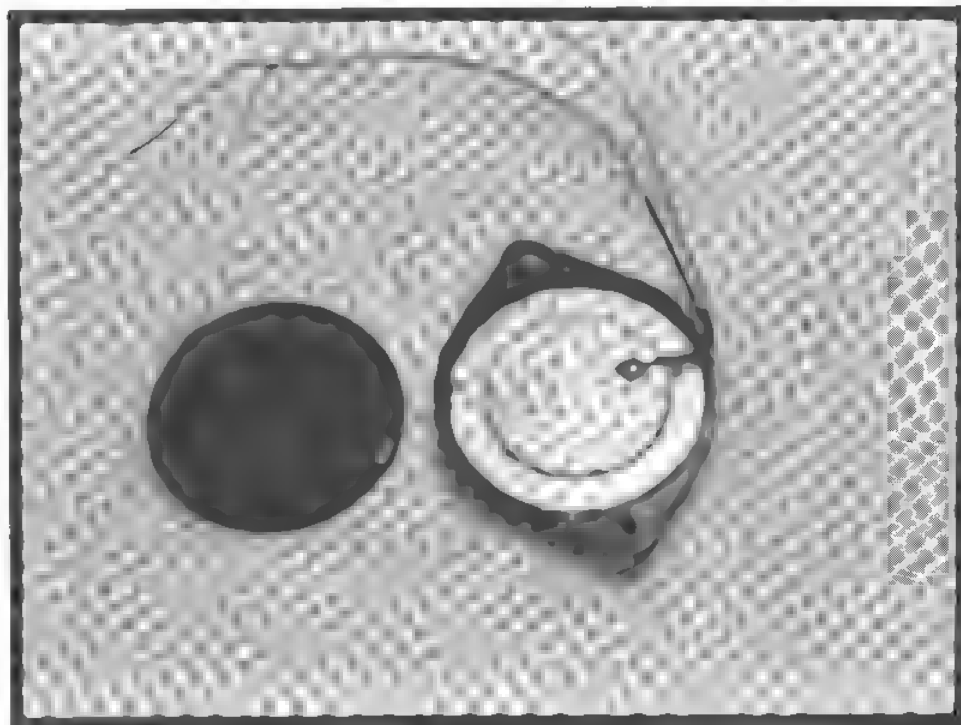
The conversion of electrical energy to mechanical energy in a piezo beeper is quite efficient—far more efficient than the energy conversion in a permanent-magnet audio speaker. This efficiency helps to conserve the battery power of the computer.

Some piezo beepers contain driving circuitry so that they can operate from direct current, yielding a fixed-frequency audio tone. In its simplest form a piezo beeper is composed of little more than the crystal and the metal disc; the beeper installed in the Model 100 is of this type. Figure 9.1 shows the beeper mounted on the inside top panel of the computer, and figure 9.2 shows the beeper fully disassembled. Wires are connected to two surfaces of the crystal.

Because the Model 100 beeper has no built-in driver circuitry, the CPU must ensure that appropriate signals are sent to it. One result is that the CPU can cause the beeper to produce a wide variety of audio signals.



**Figure 9.1.** Beeper position inside the computer



**Figure 9.2.** Beeper fully disassembled

## Hardware Theory of Operation

The beeper module, mounted on the underside of the top panel, bears reference designation P-26 and plugs into connector CN2, reference designation A-8, on the LCD printed circuit board. This is shown in figure 9.3. A flat cable runs from the LCD board to the main printed circuit board, where it is controlled by transistor T1. One side of the beeper is connected to +5 volts, while the other side goes to the transistor and a pull-down resistor.

When the transistor emitter-collector junction conducts, +5 volts is made available to the BZ line; the voltage difference seen by the beeper is small. When the transistor turns off, however, the BZ line is effectively grounded through R113; the beeper sees about 5 volts.

Transistor T1 is controlled by bits 2 and 5 of CPU output port BA or B2; (decimal 178 or 186). Table 9.1 lists the various port addresses associated with the beeper.

**Table 9.1.** Port addresses associated with beeper.

Port	Bit	Function
BA	2	Disconnects beeper from divider
BA	5	Direct beeper control line
BC	0-7	PIO divisor- lower byte
BD	0-5	PIO divisor- upper byte
BD	6-7	PIO divider mode (01=square wave, 11=pulses)
B8	7	PIO divider control (01=stop, 11=start)

The circuitry associated with bits 2 and 5 allows the CPU to control the beeper in either of two ways. With bit 2 on continually, the CPU can directly toggle the transistor and through it the beeper by means of bit 5. Alternatively, with bit 2 low and bit 5 high, the TO (timer output) signal from the PIO controls the beeper. Bit 2 can be thought of as a timer mask. Because it is one of two inputs to NAND gate (M26), a logic one at that input causes the output and the beeper to be completely unaffected by whatever the PIO timer is doing.



CPU output port BA, however, controls many functions other than the beeper. For example, setting bit 4 cuts off all electric power to the computer. Because of this, it is important not to inadvertently change any bits other than the beeper bits.

Two aspects of the PIO hardware make control of individual bits very easy. First, the present logic levels of the output port are always available to the CPU simply by reading input port BA. Second, the design of the PIO port is glitch-free. Any bits that are logically unchanged as a result of loading a new port value, will remain electrically unchanged, throughout the loading of the new data.

### CPU Toggling

Consider the simple ROM code at 7676-767C, shown in figure 9.4, which toggles bit 5. If bit 5 is on, it is turned off and vice versa.

7676	DB BA	IN BA	;IN A,(BA)
7678	EE	XRI 20	;XOR 20
767A	D3 BA	OUT BA	;OUT (BA),A
767C	C9	RET	;RET

**Figure 9.4.** ROM beeper-toggling routine

The other bits of the output port are completely unaffected. Try calling this routine from BASIC. In the immediate mode, type CALL 30326. You should hear a faint click from the beeper. Now enter and run the following BASIC program:

```
1 CALL 30326 : GOTO 1
```

You should hear a low buzz. The time interval between clicks is determined by the amount of time BASIC takes to accomplish the GOTO and parse the CALL command. If the equivalent routine were executed in machine language, the pitch would be very high, since assembly language is so fast in relation to BASIC.

One way this subroutine is used in ROM is by generating the BASIC command BEEP, which you can call from a machine language program with CALL 7662. Or, in BASIC, type CALL 30306.

Disassemble the code from 7657 to 767C. You will see the equivalent of nested FOR loops – an outer loop which determines how long the beep will last (about 117 milliseconds), an inner loop which determines how much time passes between toggling of the transistor, and the frequency, which is about 1 kilohertz. (The term “Hertz” is synonymous with “cycles per second”.)

Let’s see how these times may be calculated. The majority of the time consumed in the inner loop is in this subroutine:

```

7657      0D      DCR C      ;DEC C
7658      C2 57 76  JNZ 7657 ;JP NZ,7657

```

Register decrements require four clock cycles. Conditional jumps take seven cycles if the condition fails or ten cycles if the jump actually occurs. In this case, since C has been loaded with 50 hex (80 decimal), many decrements and jumps occur, totaling a time interval expressed as:

$$80 * (4+10 \text{ cycles}) / (2.4576 \text{ MHz})$$

This evaluates to about 0.456 milliseconds. (In this expression “cycles” refers to CPU clock cycles.)

For the sound wave emitted by the beeper to complete one cycle (one period), two toggling must occur. The audio frequency is the inverse of the period, or about:

$$(1 \text{ cycle}) / (2 * 0.456 \text{ milliseconds}) = 1100 \text{ cycles per second.}$$

In this expression, “cycles” refers to the audio signal produced by the beeper.

The duration of the beep is determined by the outer loop. The B register is loaded with zero and decremented once for each toggle until it once again equals zero. This means it is decremented 256 times. The duration is:

$$(256 \text{ toggles}) * (0.456 \text{ Msec/toggle}) = 0.117 \text{ sec.}$$

The actual BEEP frequency is somewhat lower than the calculated value. The duration is longer, because the length of the toggling period is longer than the 0.456 seconds calculated above. The subroutine at 7657 is, after all, being called by a higher routine with instructions of its own. It takes time even to accomplish the toggling.

For this application, approximate values work fine. Chapter 12 describes some activities, such as reading and writing a high-density magnetic tape, that require a careful counting of every machine cycle.

Toggling the beeper transistor is also used to monitor the cassette loading process and the Telcom carrier-detection process. Consider, for example, the code at 700D through 7011, deep in the heart of the cassette input routine:

LDA	FF44	;LD A,(FF44)
ANA	A	;AND A
CZ	7676	;CALL Z,7676

This code is reached whenever the cassette input circuitry detects a properly timed plus-to-minus or minus-to-plus transition in the incoming cassette audio data. It first inspects the contents of FF44, the location of the SOUND ON/OFF flag. Assuming SOUND is ON, the toggling routine is called.

From this you can see how SOUND ON and SOUND OFF can be accomplished in assembly language. SOUND ON is the same as loading zero to FF44, and SOUND OFF is the same as loading a nonzero value.

The audio waveform given off by the beeper is a fair copy of the waveform provided to the computer by the cassette.

A similar routine is used in TELCOM at 6EEA. The beeper is toggled in response to changes in the carrier detect bit (bit 0), of input port D8. The carrier detection filter, discussed in detail in the previous chapter, does allow noises other than a bona fide carrier tone to pass through. These noises show up in the carrier detect bit, and allow you to hear such things as a ringing phone number when you place a call to a distant modem.

It is possible to use the beeper to synthesize, albeit crudely, the human voice. Try writing a program that repeatedly samples a recorded voice played to the cassette input signal, storing in RAM the "1" or "0" that is found each time. The "1"s and "0"s are loaded to the beeper at intervals equal to the sampling intervals.

The playback interval can be varied to change the pitch of the reconstructed voice.

A fundamental rule of digital synthesis is that the number of samples per second must be more than double the desired bandwidth to be reproduced. Since the Model 100 cannot reproduce the waveform, but only the zero crossings, intelligible synthesis requires a far faster sampling rate. Synthesis of a one-second phrase might require five or eight thousand samplings.

Eight thousand samples, of course, need not fill up eight thousand bytes. Since each sample is a single binary digit, rotate instructions can be used to pack them into just 1K.

## PIO Timer Use

The PIO chip, as discussed in chapter 7, contains a divider (sometimes called a timer), which is used during UART input and output to generate the transmit and receive baud rates. The CPU crystal frequency of 4.9152 megahertz is halved by the CPU and provided for the TI (timer input) pin of the PIO. There, depending on divisor and mode data loaded on the PIO, a lower frequency can be produced at the TO (timer output).

When the UART is not in use, the divider can be connected to the beeper and loaded with a divisor to produce a desired audio frequency.

To do this in assembly language, select a divisor based on the desired frequency:

$$\text{divisor} = (2.4576 \text{ MHz}) / (\text{desired frequency}).$$

For example, to produce a concert A (=440 hertz), the divisor should be about 5585. The low-order part (5585 AND 255, which is 209) belongs in output port BC, the least-significant byte of the PIO divisor. The high-order part (5585-209)/256, which is 21, belongs in output port BD, the most significant byte of the PIO divisor.



In addition, the timer mode must be set as a “square wave”, which requires that the word sent to port BD has bit 6 on and bit 7 off. The value sent to BD is 21+64, or 85.

Next, a command word is sent to the PIO telling it to start the divider running. As described in chapter 5, this is accomplished by sending a C3 hex to output port B8. *See* table 9.1, listing the beeper output ports.

Finally, the beeper must be connected to the divider. Output port B of the PIO (CPU output port BA) needs to have bit of 2 off and bit 5 on. This can be done by reading the value of input port BA, ANDing it with FB hex (which turns off bit 2), ORing it with 20 hex (which turns on bit 5), and sending that value to output port BA. After doing all this, the beeper should sound.

A ROM subroutine, MUSIC, is available to make the beeper sound, and it is the same routine as that used by the BASIC SOUND command. The pitch is determined by the divisor in the double register DE, and the duration is determined by the byte in B; it is invoked by a CALL to 72C5. Disassemble the code at 72C5 through 7303, and use the following comments to understand it:

72C6-72C7	Send low-order byte of divisor
72C9-72CC	Send high-order byte with mode bit
72CE-72D0	Turn on divider
72D2-72D8	Connect beeper to divider
72DA-72F6	Let the tone continue, but respond to BREAK key
72F9-72FF	Disconnect beeper, resume previous beeper activity

## Musical Tones

As mentioned in the Model 100 user's manual, the BASIC SOUND command can be used to make musical tones. (Of course, from assembly language, the same routine is used by CALLing 72C5.) Unfortunately, the divisor values given there are incorrect. Figure 9.5 is a BASIC program that calculates the correct values.

The method used is simple. A concert A pitch is assumed to be 440 Hertz, although other frequencies have been used. The program could easily be modified to some other “A” frequency. For a note of any given

frequency, the note one octave above it is defined simply as the note whose frequency is double the given frequency.

The definition of the octave, by itself, does not suffice to determine the frequencies of the notes constituting the scale in between. For the last century, though, Western musicians have used a so-called equal-tempered scale. Each pair of notes, going up the scale, has the same ratio of frequencies. From this, it follows that the ratio must be the twelfth root of two, so that a change of twelve steps doubles the frequency.

Knowing this, it is easy to write a program in BASIC which calculates the frequencies and appropriate divisors.

```

5 DIM N$(11):FOR I=0 TO 11:READ N$(I):NEXT
6 DATA "A","A#","B","C","C#","D","D#","E","F","F#","G","G#"
10 FOR I=-18 TO 26
20 F=440*(2^(1/12))^I
30 D=(2.4576*10^6)/F
40 PRINT USING"      #####";N$((I+12*100)MOD12),F,D
45 SOUND D,50
50 NEXT

```

**Figure 9.5.** Calculation of divisors for notes of even-tempered scale.

The resulting tones are shown in table 9.2.

**Table 9.2.** Divisors for even-tempered tones

Note Frequency Divisor		
D#	156	15798
E	165	14911
F	175	14074
F#	185	13285
G	196	12539
G#	208	11835
A	220	11171
A#	233	10544
B	247	9952
C	262	9394
C#	277	8866
D	294	8369
D#	311	7899
E	330	7456
F	349	7037
F#	370	6642
G	392	6269
G#	415	5918
A	440	5585
A#	466	5272
B	494	4976
C	523	4697
C#	554	4433
D	587	4184
D#	622	3950
E	659	3728
F	698	3519
F#	740	3321
G	784	3135
G#	831	2959
A	880	2793
A#	932	2636
B	988	2488
C	1047	2348
C#	1109	2217
D	1175	2092
D#	1245	1975
E	1319	1864
F	1397	1759
F#	1480	1661
G	1568	1567
G#	1661	1479
A	1760	1396
A#	1865	1318
B	1976	1244

# 10

---

## The Printer Interface

---

The Model 100 communicates with a printer according to the Centronics interface standard, which defines mechanical, electrical, and software characteristics of the interface.

### **Mechanical Requirements**

The first requirement of the Centronics standard is the connector, a 36-pin device usually made by AMP or Amphenol. At the rear of the Model 100 is a 26-pin connector labeled "PRINTER", with the hardware designation CN5. This connector, with square pins spaced 1/10 inch apart, was probably chosen to save precious space on the Model 100 case. The printer cable 26-1409, the connections of which are shown in table 10.1, plugs into CN5 and has a connector at the other end that conforms to the Centronics standard.

Table 10.1. Printer connections

Centronics pin	CN5 pin	Function	Source
1	1	STROBE-not	computer
2	3	DATA0	computer
3	5	DATA1	computer
4	7	DATA2	computer
5	9	DATA3	computer
6	11	DATA4	computer
7	13	DATA5	computer
8	15	DATA6	computer
9	17	DATA7	computer
10	19	<del>ack</del> ignored by Model 100	
11	21	BUSY	printer
12	23	<del>paper</del> ignored by Model 100	
13	25	BUSY-NOT	printer
14	—	NC	
15	—	NC	
16	—	NC	
17	—	NC	
18	—	NC	
19	2	GND	
20	4	GND	
21	6	GND	
22	8	GND	
23	10	GND	
24	12	GND	
25	14	GND	
26	16	GND	
27	18	GND	
28	20	GND	
29	22	GND	
30	24	GND	
31	26	ignored by Model 100	
32	—	NC	
33	—	NC	
34	—	NC	
35	—	NC	
36	—	NC	

## Electrical Requirements

According to the Centronics standard, the thirty-six pins at the Centronics connector fall into two groups: the ground pins (pins 19 and up) and the active pins (pins 1 to 18). Ground pins are connected to a source of zero voltage in both the printer and computer. Active pins carry variable voltages, normally within the range of 0 (LOW or logic 0) to 5 volts (HIGH or logic 1). Most of the active pins are controlled by the computer and are used as a source of information for the printer. A few are controlled by the printer and are used as an information source by the computer.

The Model 100 provides ground to most of the ground pins, and makes most of the active pins accessible to the CPU. Pins 10, 12, 14 through 18, and 31 through 36 are not connected anywhere in the Model 100. For example, many printers announce that they have run out of paper by producing a high signal at pin 12. The printer cable provides this signal to the Model 100 at pin 23, but this signal goes nowhere within the Model 100.

## Software Characteristics

The Centronics standard also spells out the sequence in which the printer should energize the various lines to accomplish the printing of a character. Briefly stated, the computer determines whether the printer is able to accept another character by inspecting the BUSY and BUSY-NOT lines. The computer makes an eight-bit word available to the printer on the DATA lines, of which there are eight. It then signals the printer to read the data word by lowering the STROBE line; and leaving the data word in place long enough for the printer to read the data.

The pattern of 1's and 0's which the computer places on the eight data lines is determined largely by the ASCII character set (values 32 to 126 decimal). This is in contrast to the EBCDIC used by IBM and the Baudot code used by many telex machines. Fortunately, almost all character manipulations within the Model 100 use the ASCII set. This allows most texts to be loaded from memory to the printer without the need for translation. Most printers, however, do not respond to values between 128 and 255, let alone reproduce the novel displays the Model 100 screen gives for that range of values.

### **Model 100 Printer Hardware.**

Table 10.2 summarizes the ports through which the CPU accesses the various printer signals. The two printer status lines, BUSY and BUSY-NOT, are made available to the CPU through bits 1 and 2 of input port BB (or B3; decimal values 179 or 187). This port is implemented in hardware through port C of the PIO chip. The two lines discussed here are often referred to as PC1 and PC2. Pull-up resistors are provided, so that without a printer cable attached both appear to the CPU as logic "1".

**Table 10.2.** CPU access to printer signals.

<b>CPU source (output port)</b>	<b>Printer signal</b>	<b>CPU destination (input port)</b>
E8 bit 1  B9	STROBE DATA0 through DATA7  BUSY  BUSY-NOT	    BB bit 2 BB bit 1

The CPU sets the eight-bit data word through output port B9 (or B1; decimal values 177 or 185), implemented in hardware as port A of the PIO and often abbreviated as PA0 through PA7. Buffer M32 provides enough power to drive a two-meter cable.

The STROBE-NOT signal, usually at a logic 1 level, must be pulled down for a brief interval. This is done through bit 1 of output port E8 (actually E0 through EF, decimal 224 through 239). Port E8 is selected through port address decode line Y6-NOT and is latched in flip-flop M14. Transistor T8 is the signal driver.

These connections are shown schematically in figure 10.1.

When the printer detects the low condition of the STROBE-NOT signal, it goes to the data lines to see what character is to be printed. Obviously, the computer must leave the data lines unchanged until the printer has done this. Some computers have a parallel buffer dedicated to the printer, so that the problem never arises. In the Model 100, however, output port B9 is used for many other functions, so it is important that none of the other functions take place until the printer has had time to read the data. The designers of the Model 100 could have relied on the ACK-NOT line at Centronics pin 19, to see when the printer has read the data, but not all printers provide the ACK-NOT signal.

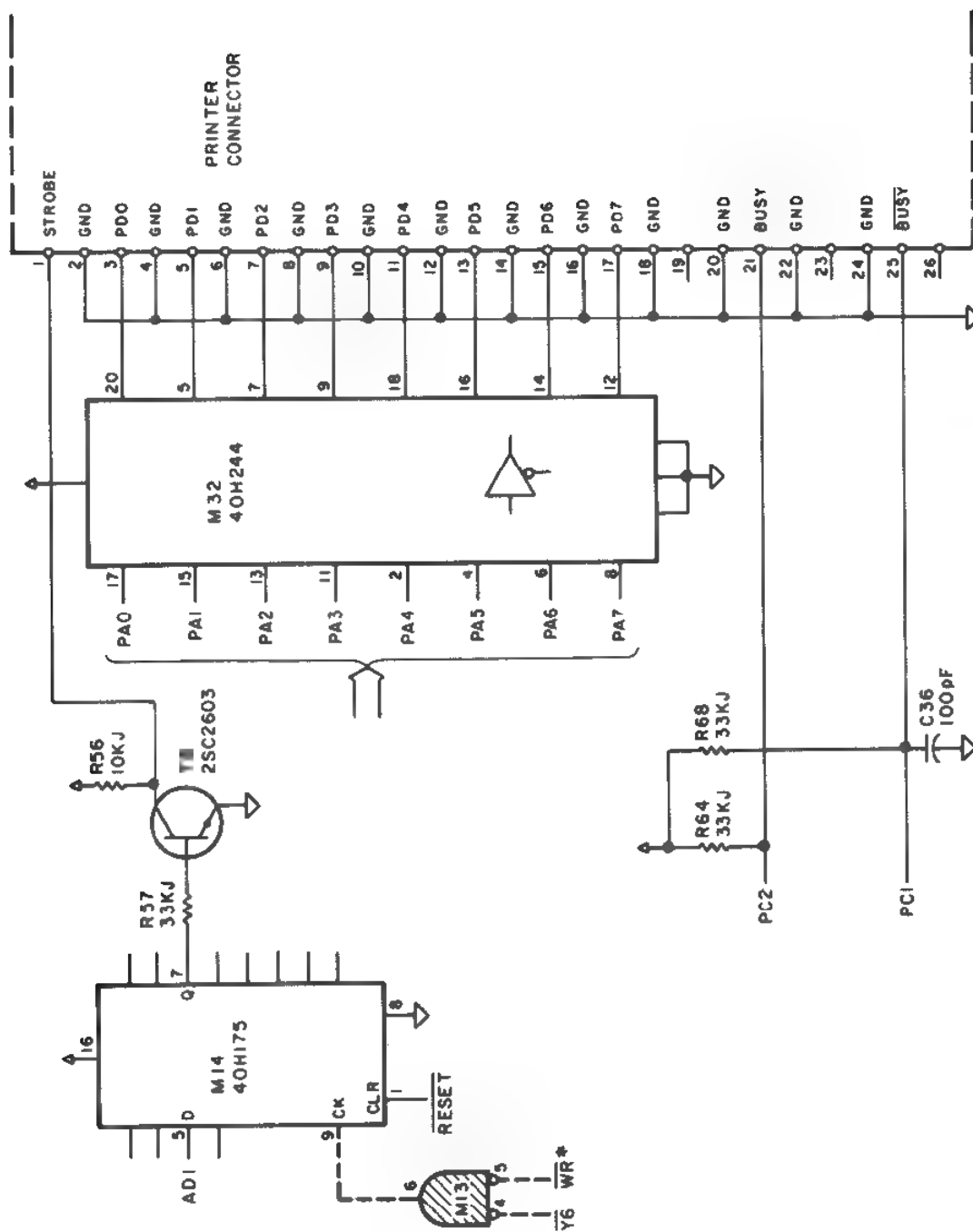
Although it might appear sensible to suspend operations until the BUSY and BUSY-NOT lines indicate that the printer has finished printing the character, with a typical 80 character-per-second printer, this would mean waiting 12 milliseconds, which is an eternity to assembly programmers. As we shall see, the ROM resolves the problem by going into a 200-microsecond delay loop (during which time most interrupts are masked) after the strobe is sent. This is based on the assumption that all printers are quick enough to read the data within that time.

## How the ROM Prints Characters

The printer driver routine is located at 6D3F through 6D6C. To understand the routine you should disassemble that code.

Since the B and C registers are used, the routine saves those registers on the stack using a PUSH instruction at 6D3F. The character to be printed, which was in A when the routine was called, is loaded to C.





### Figure 10.1. Printer hardware configuration

The CPU examines BUSY and BUSY-NOT repeatedly until the printer is ready, each time also calling subroutine 729F, which checks to see if the Break key has been pressed. If it has, the routine returns with the carry flag set. The loop is at 6D41 through 6D4D.

You want to avoid having to reset the computer if the routine is executed when no printer is attached. Model I owners will recall problems along this line.

When the printer is ready, the byte to be sent to the printer is loaded to the printer connector by means of output port B9. Then the printer strobe line is pulled low by turning on bit 1 of output port E8.

Using a single bit of output port E8 can be tricky, since it controls many other functions, such as selection of the option ROM, control of the cassette motor, and strobing of the clock/calendar chip. The Model 100 leaves a copy of the contents of output port E8 at FF45. Bit 1 is turned on and then off again, without changing the other bits as follows:

6D56	3A	45 FF	LDA	FF45	;LD A,(FF45)
6D59	47		MOV	B,A	;LD B,A
6D5A	F6	02	ORI	02	;OR 02
6D5C	D3	E8	OUT	E8	;OUT (E8),A
6D5E	78		MOV	A,B	;LD A,B
6D5F	D3	E8	OUT	E8	;OUT (E8),A

The port contents are placed in B and the accumulator, and bit 1 is turned on in the accumulator. The accumulator is sent to the port, restored to its previous value from B, and again sent to the port.

Using this ROM code, how long does the strobe last? The changes in output value of port E8 occur near the end of execution of each of the OUT instructions. They are separated by a MOV (LD) instruction that moves data from one register to another. The total time interval is fourteen cycles - the length of one OUT (ten cycles) and one LD B,A (4 cycles).

The Model 100 CPU crystal (hardware designation X2) oscillates at 4.9152 megahertz. The CPU divides this by two resulting in a clock period of about 0.4 microseconds. An interval of fourteen cycles lasts about 5.7 microseconds.

Typical printers (Epson MX-80, Okidata 92) require that the strobe last at least 0.5 microseconds, so this is more than adequate.

The code that follows, at locates 6D61-6D6C, leaves the parallel data in place during a 200 microsecond delay. It unmask the clock interrupt, restores the previous values in A, B, and C, and returns control to the calling routine.

### **Fancier print routines.**

The printer status bits available at port BB can tell much more than whether the printer is ready to receive another character. For example, with the Okidata 92 printer, the BASIC expression 6 AND INP(187) gives the following values:

- 6- If printer is not connected
- 0- If printer is connected, but the printer power is off.
- 4- If printer is out of paper or off-line.
- 2- If printer is on-line and ready.

Each model of printer, however, handles conditions such as out of paper and power off differently. In a particular application you may be able to write a program with messages like "Please turn on printer," and so on.

### **ROM Calls to the Printer**

Some ROM printer routines have been published by Radio Shack and are unlikely to change in the event of a ROM update. After calling any of these, the user should test the carry flag upon return to see if the effort to print was successful. If the carry flag is set, it means the printer hung up and the break key was pushed. The routines are discussed in turn.

### **PRINTR**

This is the routine discussed above. The character to be printed is placed in the A register and CALL 6D3F is executed. The BASIC LPOS value, however, is not updated. All register contents including A are preserved.

## **PRTLCD**

This routine dumps a copy of the LCD screen to the printer. The routine is invoked by CALL 1E5E, which is the same as the address BASIC uses when executing the keyword LCOPY.

## **PRTTAB**

This routine, invoked by CALL 4B55, sends a character in the accumulator to the printer, relying upon and updating the LPOS variable at F674. Tabs are expanded in software to spaces.

## **PNOTAB**

This routine, invoked by CALL 1470, sends a character in the accumulator to the printer, relying upon and updating the LPOS variable at F674. Tabs are not expanded to spaces but are sent as they appear. Use this routine if the printer you are sending to has tabs set in hardware at nonstandard spacings, or if the printer itself can expand tabs to spaces.

## **Printing to Dot-Addressable Graphics Printers**

Many printers have escape sequences that allow control of individual pins of a dot-matrix printing head. These sequences allow any of the 256 possible eight-bit words to be sent to the printer. Unfortunately, the printer driver invoked by the BASIC command PRINT expands every 09 hex into one or more spaces, depending on the print column position. This wreaks havoc on the escape sequences. In BASIC, the way to send such a character is CALL 5232, char where 5232 is the decimal equivalent of 1470 hex. The address is the routine PNOTAB, and char is a variable or constant to be sent to the printer.

## **Unpublished ROM Routines**

The RST 4 opcode can be used for printer output if a flag at F675 is set to a nonzero value. This is discussed in detail in chapter 13.

The routine at 4BA0 sends a carriage return to the printer, updating the LPOS variable.

### **The Low Battery light.**

When the Model 100 is turned off and the printer is turned on and connected, some printers cause the Low Battery light to turn on. A "sneak circuit" in the printer interface circuit allows this.

Referring to figure 10.1, note that transistor T8 is turned on whenever a printer strobe is desired by the CPU. This provides a ground path to the strobe line in the printer, which triggers gates in the printer to accept an incoming character.

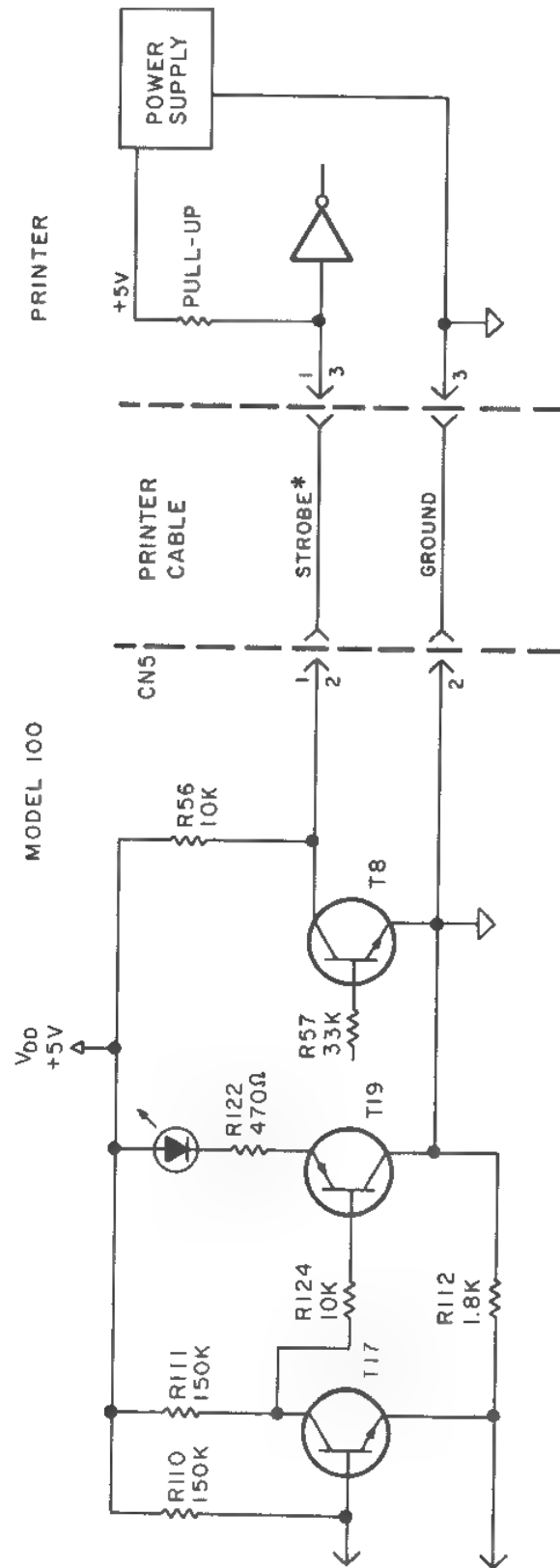
Within the printer, the strobe line must have a pull-up resistor to plus five volts. If the computer is unplugged, the printer will not start strobing in nonexistent data. The Model 100 also has a pull-up resistor, R56, which is connected to the same 5 volt source as the Low Battery light-emitting diode.

When the Model 100 is turned on and has sufficient power in the battery, transistor T17 is off. It does not turn on T19 so that the LED is kept off. If any current passes through R56, it passes toward the printer.

However, when Model 100 power is off, any weak voltage leaking back to T17, as from the printer through R56, turns T17 on. As a result, T19 turns on, and the weak voltage lights up the LED.

The best way to prevent this phenomenon is by installing a diode in series with R56, so that current can flow only toward the printer.

Not all printers do this— only the ones in which the pull-up resistor is small in resistance.



**Figure 10.2.** Connection between printer and low-battery light



# 11

---

## Clock/Calendar

---

The Model 100 keeps time and date information even when it is turned off. It does this through a sophisticated CMOS integrated circuit, the Nippon Electric uPD1990AC. This chapter discusses this chip and the associated circuitry and explains how you can use it, through ROM calls and your own routines.

### Terminology

Many devices in the Model 100 have signals and functions that can be referred to by the term *clock*. For example, the CPU provides a 2.4576-megahertz signal known as CLK to the PIO. Any output to port E8 provides four CLK signals to flip-flop M14, and bit 3 of output port B9 provides a signal called CLK to the *clock* chip, M18. We'll refer to the circuitry collectively as the *clock/calendar chip*.



## Hardware Theory of Operation

The uPD1990AC integrated circuit, designated M18 in the Model 100, is one of the chips that receives power from the AA cells or AC adapter when the ON/OFF switch SW-5 is off. This power supply, designated VB, is also supplied to the RAM chips and is backed up by the nickel-cadmium cell, so that the clock and RAM information are not lost when the AA cells are changed. The chip draws only a few tens of microamperes when keeping time.

With its crystal X1, which oscillates at 32768 hertz, it is able to keep time and date information current. The designers of the chip chose 32768 because that frequency, divided by two fifteen times, becomes one hertz and is suitable for updating the *seconds* part of its memory. The other crystals in the Model 100 are X2, which provides 4.9152 megahertz to the CPU and X3, which provides 1 megahertz to the modem chip. Neither provides a simple power of two.

The clock/calendar chip is composed of an oscillator, divider, time counter, shift register, and associated control and switching circuitry. Once it is given the values, it maintains seconds, minutes, hours, day of the week, day of the month, and month in the time counter (shown in figure 11.1). Long and short months are properly accounted for, with the exception of February 29th in a leap year.

The year is maintained not by the chip but by the ROM operating system. You may already be familiar with one bug. Occasionally the year will be incremented when it should not be; this is not the fault of the chip.

The time and date information in the time counter comprise forty bits of data. From time to time, this data must be loaded to and from the CPU. To reduce the physical size of the chip, its designers chose to use serial data transmission, which requires fewer pins than parallel transmission. The chip includes a forty-bit shift register, used for loading data into and out of the chip. Commands to the chip allow it to load serial data from the CPU into the shift register, from the shift register into the time counter, from the time counter into the shift register, or from the shift register serially to the CPU.

In addition, the chip can also be programmed to provide timing pulses (TP) of 64, 256, or 2048 hertz to the CPU interrupt RST 7.5.



The commands which may be sent to the clock/calendar chip are summarized in table 11.1.

**Table 11.1.** Clock/calendar set/read commands. The command value is placed in output port B9, then the clock is strobed by momentarily turning on bit 2 of output port E8 (or by using CALL 7383).

Command Value (hex)	Function
00	Register hold (normal timekeeping)
01	Commences shift register loading (both in and out)
02	Load shift register into time counter (set time)
03	Load time counter into shift register (read time)

The digits of time and date information are loaded into or out of the chip, as shown in table 11.2. The chip uses BCD (Binary Coded Decimal) format, in which 0000 means zero, 0001 means one, and so on, up to 1001 which means nine. Obviously BCD has much in common with hexadecimal notation; the major difference is that values like 1010 have no meaning to a machine using BCD.

**Table 11.2.** Digit sequence for chip loading

Bits	Meaning	Format
0-3	Seconds units	BCD
4-7	Seconds tens	BCD
8-11	Minutes units	BCD
12-15	Minutes tens	BCD
16-19	Hours units	BCD
20-23	Hours tens	BCD
24-27	Date units	BCD
28-31	Date tens	BCD
32-35	Day of week	0=Sunday, 6=Saturday
36-39	Month	1=January, 0CH=December

## Setting the Time in the Clock/Calendar

To set the time, the shift register must be loaded with the desired time/date information. The register is loaded serially; the serial load mode is commanded by loading output port B9 with the *shift* command 01 and then strobing the clock/calendar, as shown in figure 11.3a.

## Strobing the Clock/Calendar

Strobing is performed by turning bit 4 of output port E8 on and then off. Because the ROM operating system stores the contents of output port E8 in RAM at FF45, the proper way to strobe a clock/calendar command is as follows:

```
OUT B9 ; OUT (B9),A command to port B9
LDA FF45 ; LD A,(FF45) get contents
ORI 04 ; OR 04 turn on bit 2
OUT E8 ; OUT (E8),A strobe the chip
ANI FB ; AND F8 turn off bit 2
OUT E8 ; OUT (E8),A finish the strobe
RET ; RET wasn't that easy?
```

This routine is available at 7383 through 7390 in ROM and can be invoked by a CALL to 7383.

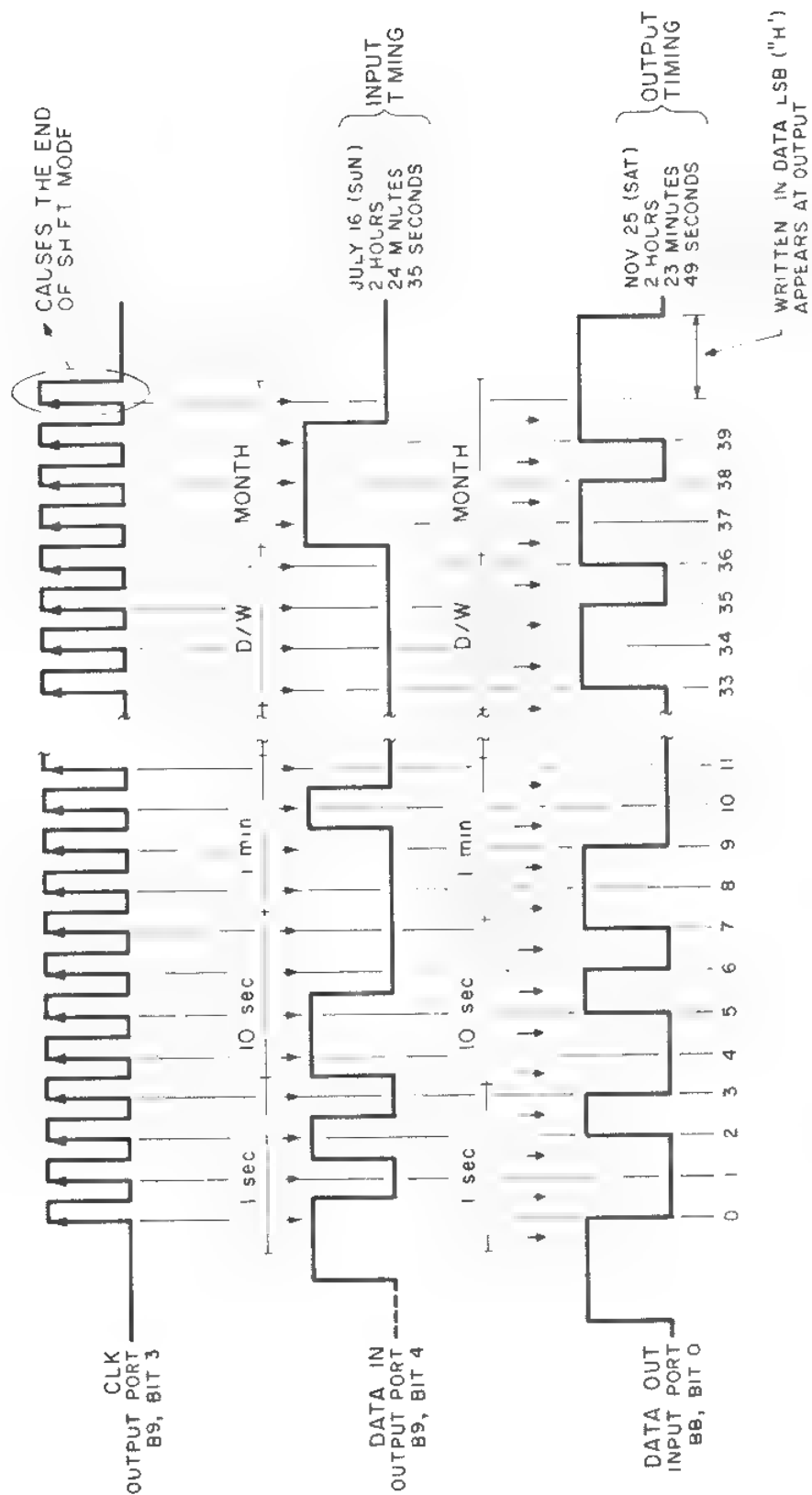
Output port B9, however, is used for many functions other than sending commands to the clock/calendar. These functions include LCD, LPT, and keyboard control. If interrupts are enabled, it is possible for the contents of output port B9 to change between the load in the beginning of the subroutine and the strobe at the fourth line. Thus, it is necessary to disable interrupts during the routine; all the ROM routines which use CALL 7383 have interrupts disabled at the time of the call.

Once the shift mode is selected, the clock/calendar chip pays close attention to bits 3 and 4 of output port B9. The contents of bit 4 (either 1 or 0) will be understood later by the chip to be bit 0 of the seconds (as shown in table 11.1).

Bit 3 of the port is then turned on and bit 4 is left in its previous state. Turning on bit 3 provides a so-called CLK input to the chip, which causes the contents of bit 4 to be loaded into the shift register. Bit 3 is then turned off. This is shown in figure 11.2.

The process is repeated for the remaining thirty-nine bits; after receiving the forty CLK inputs, the chip is no longer in shift mode. This is shown in figure 11.2.

Interrupts must be disabled during the entire process, so that the only changes to output port B9 are those selected by the main program (and not by an interrupt routine).



**Figure 11.2.** Clock calendar strobing

After the forty bits of new time/date information have been loaded to the shift register, the chip must be commanded to load the contents of the shift register into the time counter, as shown in figure 11.3b. This is accomplished by sending the 02 command, as listed in table 11.1. Finally, the chip is allowed to go back to normal timekeeping, by command 00, the register hold command.

Nothing in the hardware prevents the loading of meaningless values, such as 1111, into the shift register of the chip. Doing so would cause the chip to yield funny time values until the digit had been incremented back to zero.

### Reading the Time

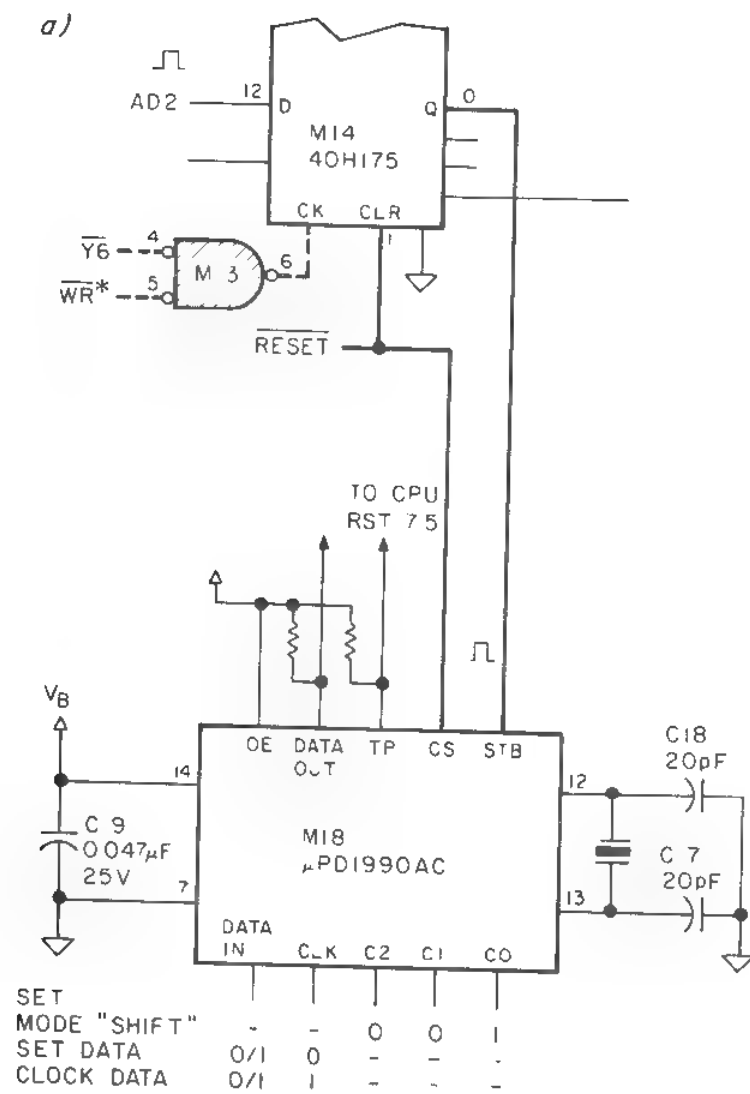
The process of reading the time is similar to setting the time. First, a read time command, 03, is strobed into the chip. This causes the time/date information to be loaded into the shift register, as shown in figure 11.4a. Then a shift mode command, 01, is strobed into the chip. The chip provides bit 0 of the shift register (the bottom bit of the seconds unit) to bit 0 of input port BB. As before, bit 3 of output port B9 gives a CLK signal to the chip. Turning bit 3 on and off again makes the shift register advance to the next bit. This is shown schematically in figure 11.4b.

After thirty-nine such loads and CLK signals (*see* the bottom plot in figure 11.2), the chip leaves the shift mode. The chip can be returned to normal timekeeping by issuing a "&0" command.

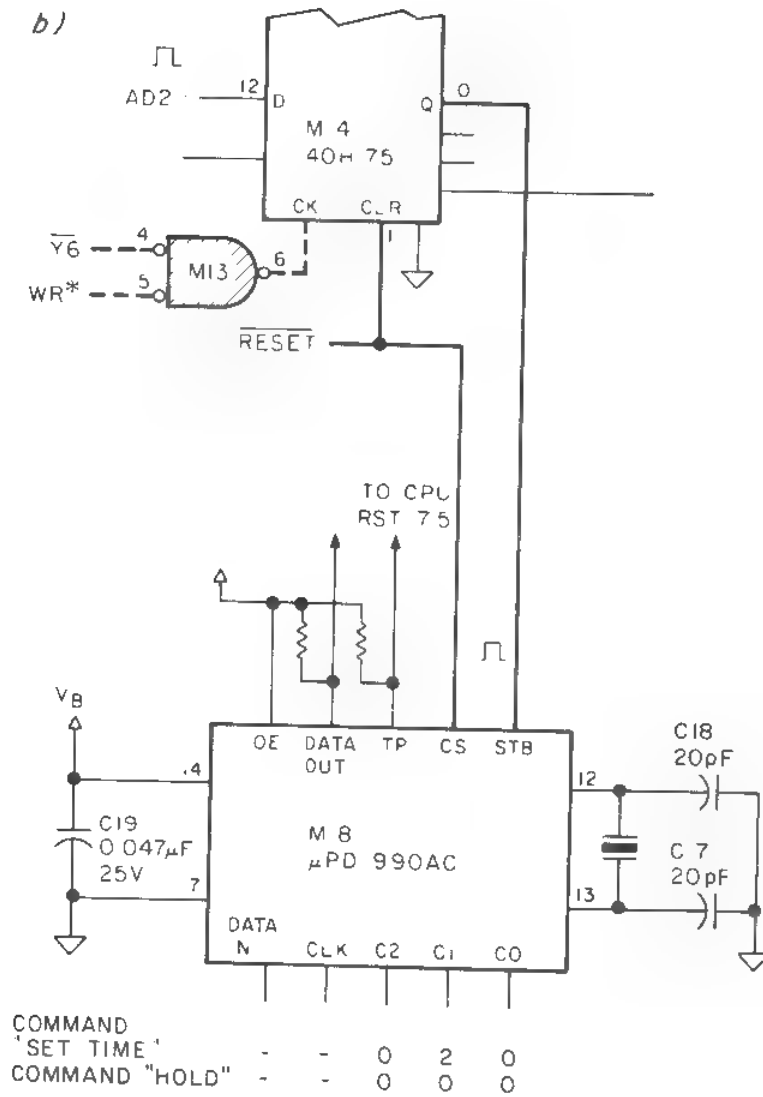
### Selecting a TP Frequency

The Model 100 ROM relies on a 256-hertz TP signal. This is provided by the clock/calendar to the CPU at its RST 7.5 interrupt input. The arrival of the TP signal at the CPU causes it to execute an interrupt routine if the RST 7.5 interrupt is enabled and unmasked. The interrupt routine performs a number of functions, including keyboard scanning (*see* chapter 15.) By placing a vector in RAM at F5FF, the interrupt can be utilized for other purposes.

In such an event, you may wish to execute the TP interrupt more or less frequently than 256 hertz. This can be accomplished by strobing a value other than 05 to the clock/calendar, as shown in table 11.3.

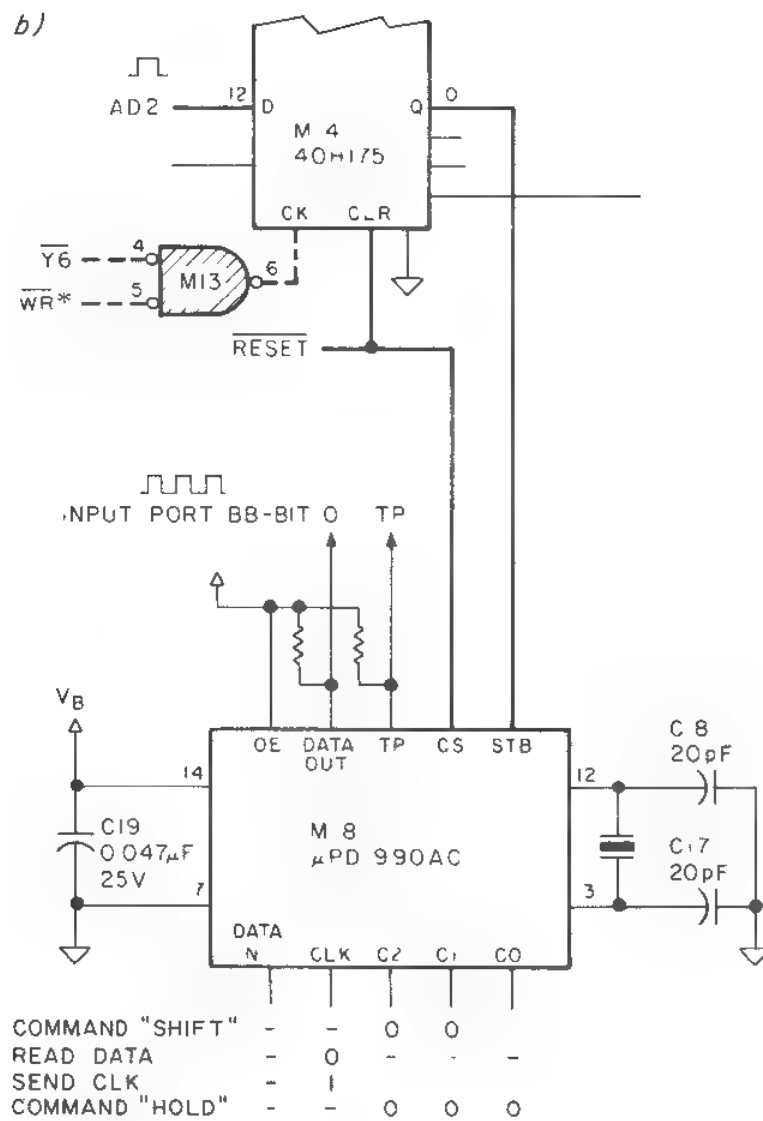


**Figure 11.3a.** Setting the clock calendar





**Figure 11.4a.** Reading the clock calendar



**Figure 11.4b.** Reading the clock calendar

**Table 11.3.** Clock/calendar TP commands.

The command value is placed in output port B9 and the clock is strobed by momentarily turning on bit 2 of output port E8 (or by using CALL 7383).

Command Value (hex)	Interrupt Frequency Selected
04	64 hertz
05	256 hertz
06	2048 hertz

The TP frequency is selected in ROM during the power-on sequence, at 6CEB.

### Clock/Calendar Accuracy

The clock/calendar keeps time as accurately as a quartz watch. The temperature of the crystal affects timekeeping; extremes of temperature throw off the time noticeably.

Most CPU functions, including enabling and disabling of interrupts, do not affect the clock. This will be a surprise to Model I/III owners, as with these computers the clock accuracy is affected by the amount of disk usage.

In particular, the power-up routines do not affect the chip which continues to keep time whether power is on or off. It is reset only by the BASIC TIME\$=, DATE\$=, and DAY\$= assignments, or by a reinitialization of RAM.

### Published ROM Routines

Three routines have been published which allow you to determine the time and date. Prior to calling any of the routines, buffer space should be set aside for the answer, and HL should be set to the starting address of the buffer to which the information will be returned. The routines are listed in table 11.4.

**Table 11.4.** Routines to obtain time and date

Data Desired	Routine Location	Format	Number of Bytes
Time	190F	hh:mm:ss	8
Date	192F	mm/dd/yy	8
Weekday	1962	ddd	3

## Unpublished Routines

A portion of the date or time information can be determined by accessing the location in RAM where the information is stored. The addresses are given in table 11.5. Before using these values, RAM should be refreshed by calling the ROM routine at 19A0.

**Table 11.5.** Date and Time locations in RAM after call to 19A0.

Values are BCD unless otherwise noted.

Address	Contents
F923	Seconds units
F924	Seconds tens
F925	Minutes units
F926	Minutes tens
F927	Hours units
F928	Hours tens
F929	Date units
F92A	Date tens
F92B	Day of week (0=Sunday, 6=Saturday)
F92C	Month (0=January, C=December)
F92D	Year units
F92E	Year tens

Note that table 11.5 resembles table 11.2. That is due to the fact that the routine called at 19A0 loads the clock/calendar shift register directly into RAM starting at F923.

Note, however, that the buffer starting at F923 is much larger than forty bits. In particular, each group of four bits from the shift register ends up in a separate eight-bit byte in RAM.

## Setting the Time through ROM Calls

The actual serial loading, both to and from the clock/calendar chip, is performed by the routine at 7329 through 7390. The routine has two entry points: 7329 to read time and 732A to set time. Prior to the call, HL must point to a ten-byte RAM area, in the format shown in table 11.6.

If the time-set entry point is used, the contents of the buffer are loaded to the clock/calendar. Only the bottom four bits of each byte are loaded to the chip, and no checks are undertaken for correctness of format. For example, 00001111 could be loaded and it would not make sense to the chip, since 1111 is not a correct BCD value.

If the time-read entry point is used, the contents of the clock/calendar are loaded to the RAM buffer. The top four bits of each byte are zero.

**Table 11.6.** Buffer for routine at 7329 and 732A

Values are BCD unless otherwise noted.

Address	Contents
HL+00	Seconds units
HL+01	Seconds tens
HL+02	Minutes units
HL+03	Minutes tens
HL+04	Hours units
HL+05	Hours tens
HL+06	Date units
HL+07	Date tens
HL+08	Day of week (0=Sunday, 6=Saturday)
HL+09	Month (0=January, C=December)

Rotations of the accumulator are used to select bits to be loaded to and from the serial input and output of the chip. The code is fascinating and well worth disassembly and study for those who wish to understand how a CPU can undertake serial input and output without a UART.

Comments are provided in figure 11.5.

7329	entry point for update of RAM
732A	entry point for update of chip
732C	disable interrupts
7331	if in RAM update mode, strobe a time read command to chip
7336	strobe a shift mode command
7339	delay forty microseconds
733E	ten digits are loaded
7340	each digit is four bits
7344	RAM update mode?
7348-D	if so, get bit 0 of input port BB
7352-B	chip update mode? if so, send a bit to port B9, bit 4
735D-63	send a CLK pulse to chip
7376	chip-set mode?
7377-9	if so, set time
737C-D	let chip return to normal
7380	reenable interrupts, return
7383-90	routine to strobe the chip with command in accumulator

**Figure 11.5.** Comments for serial clock/calendar routine

# 12

---

## Cassette Input and Output

---

The cassette interface of the Model 100 is used for storing and loading data, BASIC programs, and machine-language files.

Three file formats are used, and they correspond to the three types of RAM files: DO, BA, and CO. Data files are written on tape in 256-character blocks, each with checksum. Basic and machine language files are written in a single large (or small) block, also with checksum. A machine-language file can contain addresses for loading location and for the entry point at which execution is to begin.

## **Accessing Data (DO) Tape Files from BASIC**

Cassette data (DO) files may be easily accessed from BASIC, using the OPEN and CLOSE commands and INPUT or PRINT statements. The ROM operating system provides RAM buffers for cassette input and output, both of which are so well integrated with BASIC that one need pay no attention to the details of the tape storage format.

### **Creating a CO Cassette File**

The CSAVEM command or (SAVEM command with CAS: device specification) can be used if data is stored in memory which is to be stored on tape and later reloaded in the same memory location. The syntax is:

CSAVEM "filnam", stadd, endadd, tradd

where filnam is a filename of one to six characters, and stadd and endadd are the boundaries of the RUNM command. If no tradd value is given at the time of the CSAVEM then stadd will be used.

### **Loading a CO File back into RAM.**

When the tape file is reloaded to RAM with the CLOADM (or LOADM) command, stadd and endadd, previously written to tape with the data, are used to determine where in memory the data will be loaded.

There is one other way to access a CO file on tape. The BASIC command RUNM will load the data on the tape into RAM according to the stadd and endadd stored on tape (just like CLOADM). BASIC then commences execution of the program by JUMPing to the address stored on tape as tradd.

Often one wishes to load a tape CO file into RAM at addresses other than those stored on tape for stadd and endadd. The BASIC OPEN command is no help, because it cannot be used to access a CO tape file. The ROM routines for cassette I/O happen too quickly for

use in BASIC through CALL commands, so one is pretty much limited to assembly language for I/O involving machine-language cassette files.

### **Accessing BA Tape Files From BASIC**

You really can't do it. If you try to execute a CSAVE from within a BASIC program (as distinguished from a CSAVE executed in immediate mode), you will find that BASIC returns to the "OK" prompt when the CSAVE is executed.

This happens regardless of whether the program containing the CSAVE command has finished.

A similar problem occurs when CSAVE is used in the immediate mode. For example, if you type:

```
CSAVE"A":CSAVE"A"
```

you will find the program is saved only once.

The CSAVE command can only be used to save the BASIC program presently in the BASIC work area and cannot be used to save some other BA file in RAM memory.

The CLOAD, LOAD"CAS:" and RUN"CAS:" commands can be used within BASIC. This causes the program being run to be destroyed and the new program loaded in its place. The RUN command does result in the execution of the new program and can be used for the chaining of programs.

### **Hardware Theory of Cassette Operation**

The 8085 CPU is specially designed for CPU-controlled serial input and output. SID and SOD are dedicated chip pins used for incoming and outgoing serial data, respectively. In the Model 100 these pins are connected to the cassette interface circuitry.

Throughout the service manual the cassette circuitry bears the cryptic legend "CMT". Here the term "cassette" will be used.



## Incoming Cassette Data Flow

The cassette cord 26-1207 provides a connection to pin 4 (RXC) of the CASSETTE connector (CN3), from the earphone jack of the cassette recorder through the black plug, as shown in figure 12.1. At CN3 the signal connects with the circuitry shown in figure 12.2. At first glance it appears that diodes D5 and D6 prevent signals from passing through to the operational amplifier (op amp) M30. This is because D6 will conduct, providing a short circuit if the incoming signal is positive. If the signal is negative, diode D5 will conduct.

A diode like D5 or D6, however has the interesting property that it will never provide a short circuit. Instead, the amount of current flowing through it will be limited in such a way that its presence in the circuit never pulls the voltage across it to less than about 600 millivolts. The forward-biased voltage drop is 600 millivolts.

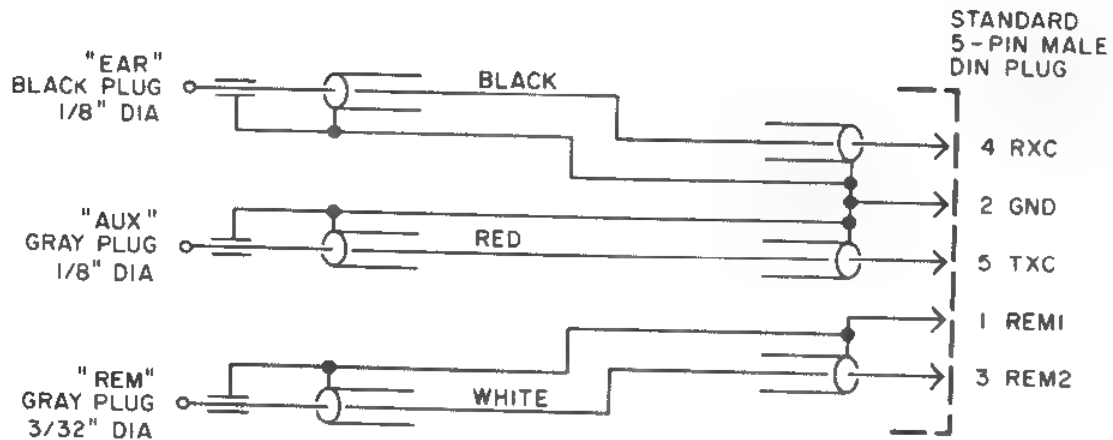
The diodes act as a limiter or "clamp" on the amplitude of the signal that reaches the op amp.

As a result, great variations in the strength of the input signal will not have much of an effect on the ability of the computer to read the data. Any signal strong enough to saturate the diodes (more than 800 millivolts) will do just fine. Increasing the volume further will not harm things either.

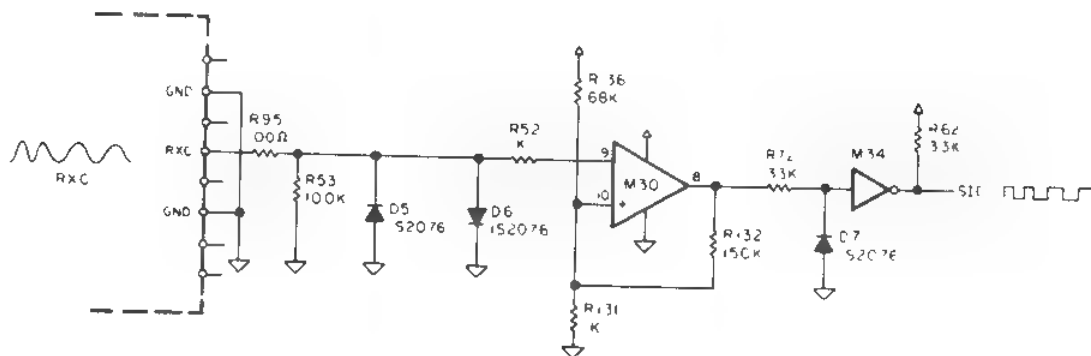
The input register R95 determines the input impedance, which is 100 ohms.

The audio signal, converted to a 600 millivolt square wave by the diodes, is amplified by op amp M30 to a square wave of about five volts magnitude. The op amp inverts the signal as the clamped audio signal is fed to the inverting input of the amplifier.

This signal is inverted (and neatly trimmed to be a "clean" square wave) by inverter M34, and is made available to the CPU at pin 5 (SID).



**Figure 12.1.** Schematic of cassette cable 26-1207



**Figure 12.2.** Incoming cassette data

## Reading the Data at the SID Terminal

The CPU reads the contents of the SID pin using the RIM (read interrupt mask) instruction. The RIM instruction and its counterpart SIM, are the two instructions added to the 8080 instruction set by the designers of the 8085.

The RIM instruction loads the Interrupt Mask to the accumulator. For cassette purposes seven of the eight bits loaded are useless, only bit 7 is meaningful. Bit 7 is a "1" if the SID pin is receiving 5 volts, or a "0" if the SID pin is receiving zero volts.

In other words, positive and negative portions of waveforms from the cassette correspond to 1's and 0's, respectively, at bit 7 of the accumulator after the RIM.

Several programming methods may be used to handle the value at bit 7. These include an AND with 80 hex or a rotate left instruction. The latter requires fewer bytes to store in ROM and is used in the Model 100 in its cassette bit input routine at 6FDB-7015.

The routine at 6FDB watches the SID pin carefully to see how much time passes between the start and midpoint of one square wave, with the result returned in the C register.

When the start of a waveform is noted, a tight loop examines the SID pin repeatedly, incrementing C until the midpoint of the waveform is seen. The loop is 29 CPU cycles long, or about 12 microseconds.

If the incoming waveform is 1200 Hertz, the time from start to midpoint would be about 416 microseconds. One would expect C to reach a value of about 35. However, if the incoming waveform is 2400 Hertz, one would expect the value of C to reach a value of about 17.

The bit-input routine has calls to 729F in each tight loop, so that pressing the SHIFT-BREAK key will allow a graceful exit from the call. Each ROM routine that calls the bit-input routine has a RC (RET C) instruction after the call for the same reason.

The bit-input routine also contains instructions that call the beeper-toggling routine at 7676 if sound is enabled according to the flag at FF44. This is how the audio monitor of cassette loading is accomplished.

## OUTGOING CASSETTE DATA FLOW

### The CPU's Role

The CPU causes cassette output by wiggling the SOD line. In ROM this is performed by the bit-output routine at 6F6A-6F84. The routine produces a single square waveform (at the SOD pin) of either 2400 Hertz (if bit 0 is one) or 1200 Hertz (if bit 0 is 0), based on the value at bit 0 of the accumulator.

The timing values in the D and E registers, set at 6F6B or 6F71, determine the time delay between the upward and downward transitions of the waveform, respectively.

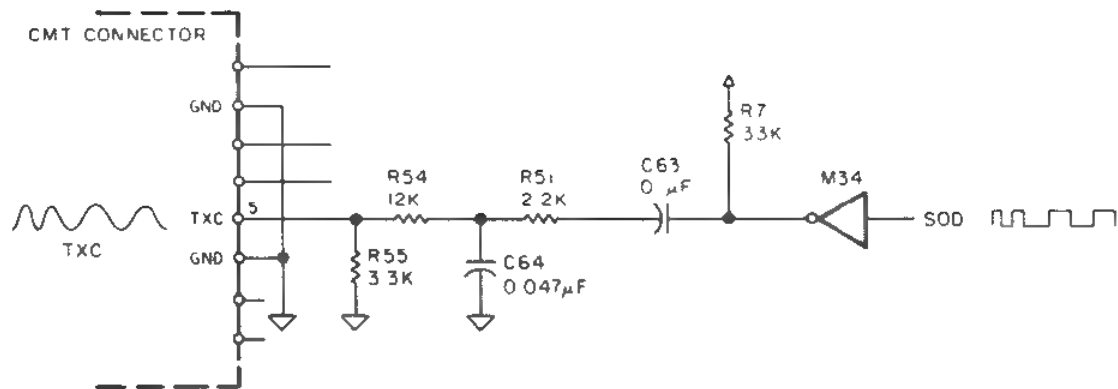
The transitions in the SOD pin are accomplished by the SIM instruction. Recall from the discussion in chapter 2 that the SIM instruction leads a double life. It sets the SOD value and masks interrupts depending on the condition of bits 6 and 3 of the accumulator prior to executing the SIM instruction. For control of the SOD pin, bit 6 should be on and bit 3 should be off. Bit 7 is loaded with the desired SOD status, either a 1 or a 0. The SIM instruction is then executed.

In simple terms, to turn SOD on, load D0 hex to A, then execute SIM. To turn SOD off, load 50 to A, then execute SIM.

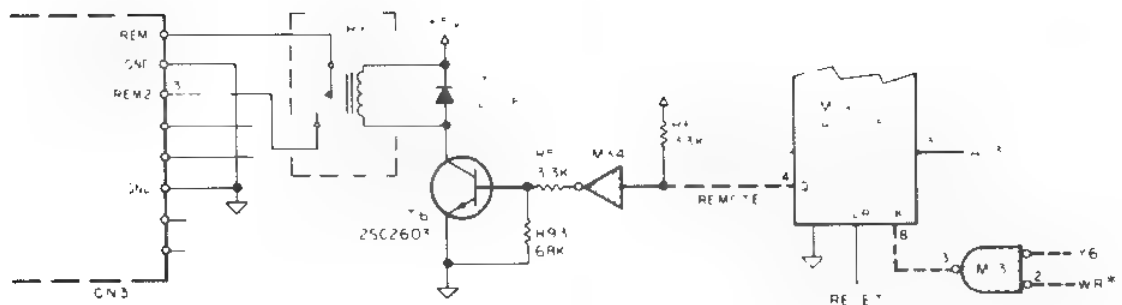
The bit-output routine performs a left-rotate. Repeated calling of the routine sends successive bits of the value in the accumulator to the cassette. This can be seen in the routine DATAW at 6F5B-6F67, which writes to tape the character in the A register. It simply sends one 1200-Hertz cycle, then calls the bit-output routine eight times, sending eight square waves, each of which may be 1200 Hertz or 2400 Hertz.

### Hardware Treatment of the SOD Signal

If you try to record a square wave and play it back, the result will no longer be a square wave. This is because the sharp corners produce unwanted signals (harmonics, in the language of Fourier analysis) as the wave passes through the system. To avoid this, the square wave emitted from the SOD pin is passed through a resistor-capacitor network designed to smooth the wave, as shown in figure 12.3.



**Figure 12.3.** Outgoing cassette data



**Figure 12.4.** Cassette motor control

The recording level in most recorders cannot be adjusted and is instead determined by an ALC (automatic level control) circuit which adjusts the recording level according to the loudness of the incoming signal.

The ALC circuit requires a fraction of a second to stabilize when encountering a signal after a period of silence. Therefore it is good practice to write a few cycles of "leader" before each stretch of data.

## **Interrupts**

Obviously any routine, whether a routine from ROM or one you write yourself, which is designed to read or write cassette data, must have interrupts disabled. If interrupts were handled the timing loops would be thrown off.

## **Motor Control**

The CPU turns the cassette motor on and off by way of bit 3 of output port E8. As that port controls other functions, including the option ROM, the printer strobe, and the clock/calendar strobe, it is important to change only bit 3 when it is desired to turn the motor on or off. The ROM routines store the current contents of the port in FF45, and you should do this too.

The contents of FF45 are ORed with 08 to turn bit 3 on and ANDed with F7 to turn bit 3 off.

Once the bit is set, the accumulator may be loaded to port E8 and also to FF45.

As shown in figure 12.4, the value at bit 3 of the output port drives a flip-flop M14 to produce a signal called REMOTE\*, which is low when the Motor is on and high when the motor is off. Remote\* is inverted by M34 so that a "1" at bit 3 turns the transistor on. Turning the transistor on provides a ground path for the coil of relay RY1. Energizing RY1 closes the contacts (which are normally open) which shorts pins 1 and 3 of the CASSETTE connector CN3.

These two pins, designated REM1 and REM2 in the Model 100, are connected by the cord to the 3/32" gray plug, as shown in figure 12.1. That plug is intended to go in the "REM" jack of the recorder, which turns the motor on and off. (Obviously the REM1/REM2 control capability may be used for control of other things, as long as the current is not too great.)

The one place in ROM where bit 3 of output port E8 is controlled is at 740F. The routine there is jumped to by the motor-off and motor-on subroutines discussed below.

### **Published ROM Subroutine Calls**

Before cassette data is read or written, interrupts must be disabled and the motor turned on. Both can be accomplished by a call to CTON at 14A8.

The motor is turned off, and the interrupts are enabled by a call to CTOFF at 14AA.

### **Writing to Cassette**

The routine DATAW, called at 6F5B, writes the byte of data in the accumulator to tape. (This routine is discussed above). If the SHIFT-BREAK key is pushed, the routine will return with the carry flag set.

The routine CSOUT, called at 14C1, sends a character to the cassette, keeping track of a checksum. Prior to the call, the byte to send is in the accumulator and the current checksum is in C. After the call, the updated checksum is in C. (This routine calls DATAW, so it also returns with the carry flag set if the SHIFT-BREAK key was pushed.)

By "checksum" we simply mean a running total of bytes sent to tape. The values are added, ignoring the carry bit.

The routine SYNCW, called at 6F46, writes a header and sync byte to cassette. The header is composed of 512 bytes, each 01010101, all run together end to end. This is the steady tone you hear at the beginning of any cassette load. The sync byte is a 01111111, which is a warning to the cassette input routine that the 01's are almost at an end, and that what follows is real data.

This routine will exit with the carry flag set if the SHIFT-BREAK key is pushed.

## Reading From Cassette

A single byte may be read with DATAR, called at 702A. DATAR uses the bit-input routine discussed above. Upon return from the bit-input routine, DATAR examines the C register to see how long the waveform was. The threshold applied is 21 decimal or about 2000 Hertz. The 1 or 0 is shifted, and the bit-input routine is called again until eight bits have been received (or the SHIFT-BREAK key pushed).

The result of the read appears in the D register. Like all the “read” routines, it checks to see if the SHIFT—BREAK key was pushed. DATAR responds to that event by returning with the carry flag set.

The routine CASIN, called at 14B0, reads in a character by calling DATAR. If the SHIFT-BREAK key was pushed, the routine ends by turning the motor off and jumping to the BASIC routine that displays the “10” error message on the screen. (If you don’t want this response to a SHIFT-BREAK, you can adopt the code from 14B0 to 14C0).

The routine will update the checksum value residing in the C register.

The routine corresponding to SYNCW above is SYNCR. This clever routine, called at 6F85 and located in ROM at 6F85-7042, listens to the tape until it finds that the incoming signal is between about 660 and 5300 Hertz, figures out where the 01’s are, and then waits for the sync byte (the 7F mentioned above). At that point, or when the SHIFT-BREAK key is pushed, it returns.

## Unpublished Routines

The routine at 22B9 sends DE bytes to cassette, from a buffer pointed to by HL. This is followed by a checksum designed to add up with the previous bytes to make a total of zero, followed by twenty 0 bytes. The routine then turns the motor off and jumps to 0501. Since it never returns, it is unsuitable for assembly language use. It may, however, be adapted to make a good block-write routine.

The routine at 2413 will load up to 256 bytes from cassette into the RAM buffer pointed to by HL. The DE register-pair contains the number of bytes to load. For 256 bytes, the register should contain 00. When the routine finishes, the checksum for these bytes is in A.



By convention blocks of data are written so that the checksum is zero. When the routine is finished, if the checksum is zero, the Z flag is set, representing a successful load.

## File Formats

Cassette files are composed of two or more blocks, each of which starts with a header and a sync byte.

The first block in the file contains a header (512 bytes of 55H), a sync byte (7F), the file type (9C for "DO" files, D0 for "CO" files, D3 for "BA" files), the filename (six bytes), and other information (10 bytes).

The second and subsequent blocks are composed of a header (512 bytes of 55H), a sync byte (7F), a data indicator (8D), and then the data followed by a checksum. DO files are broken up into 256-byte blocks, while BA and CO files have one enormous block for the whole file contents.

A file may be opened using the subroutines listed in table 12.1. In each case, prior to the call, the filename must be stored at FC93. Other data, if any, is stored at FACE.

**Table 12.1.** Call locations for file OPEN

File Type	Symbol	Open for Output	Open for Input
BA	D3	260B	2650
CO	D0	2611	2656
DO	9C	260E	2653

## User Experimentation

Nothing in the hardware requires you to use frequencies of 1200 and 2400 Hertz to store data. Nor is there any requirement that the storage technique be single square waves of particular wavelengths. Feel free to modify the ROM bit-input and bit-output routines, or write your own routines for other formats.

For an example of another format, see MCS-80, 85 Family User's Manual, pages A1-38 to A1-42. There, bursts of tone and periods of silence of varying duration are used to represent 1's and 0's.

It is also possible to vary the format of data within a cassette file. It is not necessary to use the BA, CO, or DO cassette file formats if you are willing to design your own format.



# 13

---

## The Liquid-Crystal Display Screen

---

The Model 100 top panel contains a screen composed of a rectangular array of so-called “pixels”, 240 across and 64 down. Each pixel is about 0.8 millimeters square. The pixels are part of what is called a liquid crystal display.

### **How Liquid Crystals Work**

If you hold two polarizing filters between your eye and a table lamp, you will find that the amount of light that passes through is a function of the angle between the two filters.

One easy way to try this is to obtain two sets of Polaroid™ sunglasses. Hold one pair with the lenses side by side and the other pair with the lenses one above the other. No light will pass through. If the glasses are held with the lenses side by side, light will pass through.

This is due to the fact that light can be polarized. When light passes from the lamp toward you and through the furthest polarizing lens, it has been filtered so that all the light is polarized. Let's assume it is polarized vertically. When this vertically polarized light reaches the lens closest to your eye, it will pass through only if the lens is turned the same way as the first lens (so that it also passes vertically polarized light).

Some transparent substances have no effect on light passing through them, while others will twist the polarization of light passing through them. Depending on the particular substance and the distance travelled through the substance, vertically polarized light may be converted to horizontally polarized light, and vice versa.

Such substances are not hard to find. Everyday dextrose, a common sugar made from corn, will twist polarized light. Its very name was chosen because it twists light to the right. "Dextro-" is a prefix which means "to the right".

A liquid-crystal display is composed of a carefully prepared liquid placed between two glass panels. The top panel includes a polarizing filter; the top and bottom panels contain nearly transparent electrodes extending vertically and horizontally.

Room light striking the screen passes first through the polarizing panel. The furthest penetrating light is the light which is polarized, say, vertically. This light bounces off the bottom panel, and (if the polarization has not changed) passes through the polarizing panel a second time, reaching the eye. Any part of the screen where this occurs is perceived as being light in color.

The liquid is designed to twist the light when it (the liquid) is subjected to an electric field. For a certain pixel to be perceived as dark, the LCD driving circuitry must activate the row and column electrodes associated with that pixel every so often. This occurs once every fourteen milliseconds and lasts about half a millisecond.

The electric field being emitted from the electrodes causes the liquid to twist the light a certain number of degrees.

The eye's viewing angle has an effect on the amount of polarization associated with the glass panels, so that no single number of degrees of twist will produce a dark panel for all viewing angles. The rotary control DISP on the right side of the Model 100 is a potentiometer, which varies the voltage used to activate the pixels and optimizes the appearance of the screen for a particular viewing angle.

## **CPU Control of the Screen**

CPU control of the 15360 pixels is accomplished through ports FE (LCD data) and FF (LCD status/command), as well as output ports B9 and BA. Specialized integrated circuits (HD44102 and HD44103) are used to drive the pixels, and provide the 15360 bits of RAM memory needed for LCD operation. (A detailed discussion of LCD I/O ports is beyond the scope of this book. For further information, see the Model 100 Service Manual pages 4-13, 4-14, 4-28, 4-29, 4-30, 7-1, and 7-2.)

The 15360 bit RAM memory mounted on the LCD printed circuit board is not directly addressed by the CPU. Instead, it is loaded through the I/O ports. Some of the regular RAM memory, located from 8000 to FFFF, however is allocated to LCD data. The area from FE00 to FF40, for example, contains the ASCII values presently on the screen. This is the source of information used when BASIC performs the LCOPY command.

It is clear however, that the actual screen contents (the on/off states of the pixels) are not found in RAM at FE00-FF40. For example, if PSET and PRESET are used to turn pixels on and off, the LCOPY command will not convey the results to the printer, even if the pixels form a printable character. Similarly, the patterns that reach the LCD by means of PSET and PRESET will not scroll upwards when the rest of the display does.

## **Character Formation**

When sending data to the LCD screen, the CPU does not send ASCII values to the integrated circuits on the LCD board. Instead, it sends 1's and 0's which are to be stored in the LCD RAM. The LCD RAM is used to drive the individual pixels.

The ROM routines used by the CPU in handling screen output use several different routes for the data. In the case of pixel-specific routines like PLOT and UNPLOT (used in the BASIC commands PSET, PRESET, and LINE) the ROM routine sends addresses and data to the LCD chips to affect only the pixels in question.

When ASCII characters are printed to the screen, the routine first loads the ASCII value to the RAM area FE00-FF40. It later interprets the ASCII value according to a ROM table to determine which bits must be turned on and off to form that character.

### Formation of Character Shapes

No character shape information is needed for ASCII values from 0 to 31 (decimal) as these are not printable characters. They instead merely cause cursor movement, etc.

Since each character printed on the screen lies in an array that is six pixels wide and eight pixels deep, forty-eight bits of data are required to define the whole character. (It happens that the rightmost column is always empty in the case of ASCII values 32 to 127. As we shall see, the ROM storage technique takes advantage of this fact to save ninety-six bytes of ROM.)

The character-generation table begins at 7711 and runs to 7BF0. (The ROM routine that uses it is located at 73EE). To see how the table works, print out (using the PEEK function) the five values starting at memory address 78CE (30926 decimal). The contents of these locations are 28, 160, 160, 144, and 124. Now, convert each of these numbers to binary notation. The results are 000111000, 10100000, 10100000, 10010000, and 01111100. If these binary numbers are written in a column, something interesting will emerge, as shown in figure 13.1.

30926	00011100
30927	10100000
30928	10100000
30929	10010000
30930	01111100

**Figure 13.1.** Lower-case “y”

Do you see it? Turn the page sideways, and you will see a lower-case "y" among the 1's and 0's.

The table begins at 7711 with the pixel information for an ASCII 32 (decimal) which is a space. As you would imagine, it is composed of all zeros. The table continues, five bytes at a time, through ASCII values 33 to 127.

At location 78F1 the table changes. Since many of the characters beyond 127 use all six columns of pixels, six bytes of data are used for each character. The character with value 128 (which looks a little like a telephone) occupies 78F1, 78F2, 78F3, 78F4, 78F5, and 78F6. From this point to the end of the table, each character uses six bytes. The table finishes at 7BEB-7BF0 for the character with value 255.

It is interesting to use this ROM table to generate the screen characters yourself. The program in figure 13.2 prints the 224 printable characters of the Model 100 to the screen. Each character is displayed twice — once in the normal way by use of the BASIC PRINT command, and a second time with pixels turned on one by one to form the characters.

The two images of the character are identical in appearance because they are both based on the bit-graphics information in the ROM table. What's different is the sort of programming that puts the bits on the screen. When the PRINT command is executed, BASIC invokes machine-language subroutines which extract the information from the ROM table and put it on the LCD screen — all the pixels turn on, forming the character, virtually simultaneously. The second character image reaches the screen much more slowly (you can see that the pixels turn on one by one) because the calculation of which pixels to turn on is done step-by-step in BASIC.

The BASIC PRINT statement gives the ASCII value to an assembly language routine in ROM, which uses the machine language subroutines PLOT and UNPLOT to turn on the proper pixels.



```
10 CLS:FOR CR= 32 TO 255: PRINT @129, CR;
   :PRINT@134, CHR$ (CR);:IFAS < 128THEN
   AD=30481+(CR-32)*5
   ELSEAD=30961+(CR-128)*6
14 FOR COL=0 TO 5: BY=PEEK(AD+COL)
   :IFCR < 128 AND COL=5 THEN BY=0
16 FOR ROW =0 TO 8:IF BY AND (2 ^ROW)
   THEN PSET(96+COL,24+ROW)
   ELSE PRESET(96+COL,24+ROW)
20 NEXT ROW:NEXT COL:BEEP
100 IF INKEY$="" THEN 100
   ELSE NEXT CR:END
```

**Figure 13.2.** Program to demonstrate the ROM character-generation table.

If your printer includes bit-addressable graphics, such as the Epson MX-80 with Graftrax, you can print the CODE and GRPH characters directly at the printer. A program to print the values in table 13.1 is shown in figure 13.3.

```
10 kl=31729 : e$=chr$(27)+chr$(75)+chr$(6)+chr$(0)
   : for as=32 to 127:for kt=0 to 43: if peek(kl+kt)
   <>as then 1000 else for co=0 to 5:
   va=peek(kl+kt+co*44) :if va =0 then lprint space $(12);
   :goto 900
20 lprintusing"    ### "
   ;va;:lprinte$;:ad=5*va+30321:
   if va>127 then ad=va*6+30193
30 for bi=0 to 4: a=peek (ad+bi) :gosub 2000
   :next bi: if va>127 then a=peek(ad+5)
   :gosub2000 else a=0:gosub 2000
900 next co:lprint
1000 next kt:next as:end
2000 a1=0:for ib=0 to 7:
   a1=a1 or ((a and 2 (7-ib))<>0) and 2 ib)
   :next ib: call 5232, a1:return
```

**Figure 13.3.** Program to print Model 100 characters to bit-addressable dot-matrix printer.

Let's analyze the program line by line and see how the printing is accomplished.

Line 10 sets up a FOR loop which picks an ASCII value and searches the keyboard-decoding ROM table (*see* chapter 6) for the place in the table where that lower-case key is located.

When a particular lower-case key is found, the uppercase GRPH SHIFT-GRPH, CODE, and SHIFT—CODE ASCII equivalents are extracted from the keyboard-decode table through the expression:

```
va=peek(kl+kt+co*44)
```

The resulting six ASCII numerical values are printed by lines 20, 30, and 2000. In a few cases there is no ASCII value. For example, no value is assigned to CODE-G. In such cases the program simply prints twelve spaces.

The Epson printer with Grafrax uses escape sequences to output the bit-addressable graphics. The sequence is an escape (decimal 27), the letter N (decimal 75), the number of columns to be printed bit-style (decimal 6), and a null (decimal 0). The next six values received by the printer are generated much like the array in figure 13.1. Each "1" in binary notation results in a dot on the paper from the print head.

Unfortunately, the Grafrax protocol assigns the bits to the paper "upside-down" from the way the Model 100 assigns the bits to the screen. The FOR loop of line 200 inverts the bits before printing.

The BASIC PRINT routine converts any ASCII TAB (decimal value 9) to a varying number of spaces based on where the Model 100 thinks the printer carriage is positioned on the printer. This is handy for Radio Shack printers that don't know where the next tab stop is, but can cause problems when you want to send escape sequences which sometimes contain the value "9".

The PRINT routine can be circumvented with a machine-language subroutine call as shown on line 2000.

Unshifted	SHIFTed	GRPH	SHIFT-GRPH	CODE	SHIFT-CODE
39	34	140		160	164
44	60	157	248	188	221
45	95	92	124	197	167
46	62	151	247	207	
47	63	138		174	
48	41	125		175	166
49	37	136	225	192	208
50	64	156	226		
51	35	157	227	193	209
52	36	158	228		
53	37	159	229		
54	94	180	230		
55	78	176		196	212
56	42	163		194	210
57	40	123		195	211
58	58	146	245	173	
61	43	141		190	168
91	97	96	126	181	
97	65	133	235	182	177
98	66	149			
99	67	132	255	162	171
100	68		237	187	215
101	69	147	273	198	214
102	70	170	238		191
107	71		253		
104	72	134	251		
105	77	142	247	199	213
106	74		244	203	219
107	75	155	250	201	217
108	76	154	249	202	218
109	77	129	246		165
110	78	150		205	
111	79	152	242	183	178
112	80	128	241	172	
117	81	147	231	200	216
114	82	137	234		170
115	83	139	236	169	185
116	84	135	252		186
117	85	145	240	184	179
118	86			189	222
119	87	148	232		
120	88	171	239	161	223
121	89	144	254	204	220
122	90		224	206	

Table 13.1. LCD characters

## RAM Locations Relating to the Display

A number of RAM locations are set up when the Model 100 is initialized, and should be left undisturbed, as should everything above F5F0, by any user program. The most commonly used values are listed in table 13.2.

**Table 13.2.** Display variables in RAM

Name	Address	Description
CSRY	F639	Horizontal cursor position
CSRX	F63A	Vertical cursor position
	F63B	Number of active cursor lines
	F63C	Number of active cursor lines
	F63D	Line-8 lock flag
	F63E	Scrolling disable flag
	F648	Reverse "video" flag
	F675	Output flag 0=LCD 1=LPT
	F788	BASIC POS value
	FCC0	Beginning of alternate LCD buffer
	FDFF	End of alternate LCD buffer
BEGLCD	FE00	Beginning of LCD memory
ENDLCD	FF40	End of LCD memory

## Published ROM Subroutine Calls

The most frequently used ROM call is LCD, at 4B44. The character in the accumulator is put on the LCD screen at the current cursor position, and the cursor moves to the right (and if necessary, to the next line). This routine is somewhat like the Model I/III routine VDCHAR at 0033. Assuming scrolling has been enabled, then scrolling will occur if necessary.

The LCD routine is quite versatile. While no one would be surprised at its response to printable ASCII values (decimal 32 and above), the routine also handles certain values less than 32. These values are shown in table 13.3.

**Table 13.3.** Nonprintable values which may not be sent to the LCD routine

Value	ASCII Meaning	Call to Send	ROM Address	Response
07	Bell	4229	7662	Beeping sound
08	Backspace		4461	Moves cursor to left
09	Horizontal tab		4480	Moves to next tab column- 8,16, etc.
0A	Line feed	4225	4494	Line feed-column remains the same
0B	Vertical tab	422D	44A8	Home cursor
0C	Form feed	4231	4548	Clear screen & home
0D	Carriage return		44AA	Cursor to left edge-row does not change
1B	Escape		43B2	Interpret next character

The nonprinting values are decoded according to a ROM table at 438A-43A1. The escape sequences, in turn, are decoded in a ROM table at 43B8-43F9. The addresses of the routines to accomplish the various escape sequences can be determined from the ROM table. There are twenty-one permissible LCD escape sequences which are listed in table 13.4.

**Table 13.4.** LCD Escape Sequences.

Name	Call	Hex	Chr	ROM Address	Function
		41	A	4469	Up one line unless already at edge
		42	B	446E	Down one line unless already at edge
		43	C	4453	Right one space unless already at edge
		44	D	445C	Left one space unless already at edge
		45	E	4548	Same as printing 0C-clears screen
		48	H	44A8	Same as printing 0B-moves cursor to 1,1

*continued on following page*

		4A	J	454E	Erase from cursor to end of line
ERAEOL	425D	4B	K	4537	Erase from cursor to end of line
INSLIN	4258	4C	L	44EA	Insert a blank line on LCD at cursor
DELLIN	4253	4D	M	44C4	Delete a line on LCD at current line
CURSON	4249	50	P	44AF	Turn on cursor
CUROFF	424E	51	Q	44BA	Turn off cursor
SETSYS	4235	54	T	4439	Set system line (lock LCD line 8)
RSTSYS	423A	55	U	4437	Reset system line (unlock LCD line 8)
LOCK	423F	56	V*	443F	Lock LCD display (no scrolling)
UNLOCK	4244	57	W	4440	Unlock LCD display (allow scrolling)
	4262	58	X	444A	Repaint screen
		59	Y	43AF	Cursor position (see text)
		6A	J	4548	Same as printing 0C-clears-screen
		6C	1	4535	Erase entire line containing cursor
ENTREV	4269	70	p	4431	Set reverse character mode
EXTREV	426E	71	q	4432	Turn off reverse character mode

In other words, if a character value listed in table 13.3 is "sent to the screen" by the LCD routine, the action shown in the table will be taken. If the character "sent to the screen" is an ASCII escape character (decimal 27) then the character that follows will be interpreted as an escape sequence as shown in table 13.4 and not as a printable character.

---

\* Radio Shack incorrectly lists this as ESC Y.

Several of these routines are used directly by BASIC. The BASIC command CLS is executed by means of a call to the CLS routine at 4231. (CLS is equivalent to the Model I/III routine VDCLS at 01C9.) Also, note that the BASIC command BEEP is executed by means of a call to the routine at 4229 listed in table 13.3.

## How to Send Special Characters

There are several ways to program each of the functions listed in these tables. The first is simply to load the character (or characters, in the case of an escape sequence) into the A register, and execute one or more RST 4 instructions. This requires several opcodes to execute however.

Alternatively, you can call the address listed as "CALL address" in the table. If you disassemble that code, you will find that in each case the value is loaded to the accumulator, and the RST 4 is invoked. Because these call addresses have been published by Radio Shack, they are likely to survive any ROM upgrades.

Another means of programming the functions, in the case of the escape sequences, is to use the ESCA subroutine at 4270. Before calling the routine, place the value of the desired escape sequence from table 13.4 in the accumulator.

Finally, it is possible in each case to directly call the routine shown in the "ROM address" column. The advantage is faster execution time, while the disadvantage is that the address may change with a ROM upgrade.

Sending a carriage-return-line-feed combination to the screen can be accomplished with a call to CRLF at 4222 as shown below:

4222	3E	0D	MVI	A,0D	;LD A, 0D
4224	E7		RSI	4	; send to LCD
4225	3E	0A	MVI	A,0A	;LD A, 0A
4227	E7		RST	4	
4228	C9		RET		

This routine will save four bytes each time you use it.

## Sending Characters to the Printer

The LCD routine is versatile in other ways. It relies on a flag stored at F675 which, if zero, indicates that output should be directed to the LCD, as the name suggests. If the value at F675 is nonzero, the value in the accumulator will be sent to the line printer instead. This may be seen, for instance, in the code for LLIST at 113B and the code for LIST at 1140:

```

113B  3E  01          MVI A,01    ;LD A, 01
113D  32  75  F6     STA F675    ;LD (F675), A
1140  C1              POP B      ;beginning of LIST routine

```

To send output to the printer, simply set the printer flag at F675 before calling RST 4.

## How to Call 4B44

If you disassemble all of ROM, you will see that LCD is never invoked by a CALL 4B44. Instead the ROM designers placed a JP 4B44 at ROM address 0020, so that an RST 4 (sometimes called an RST 20) opcode may be used, saving two bytes of ROM each time it is called.

Obviously, not all uses of the LCD routine may be accomplished by an RST 4. For example, a conditional call cannot be accomplished in less than three bytes. This may be seen in ROM at 4B3F and at 54BC.

## Other Published LCD ROM Routines

Two routines allow the machine language equivalent of the BASIC commands PSET and PRESET (which are abbreviations of "pixel set" and "pixel reset"). These are PLOT at 744C and UNPLOT at 744D, respectively. In each case the pixel to be changed is addressed through the DE register-pair. D contains the X coordinate between 0 and 239. E contains the Y coordinate between 0 and 63.



## Cursor Position Routines

The routine, POSIT allows a machine language program to handle the cursor directly. POSIT, at 427C, moves the cursor to the position given in the H (column 1-40) and L (row 1-8) registers. This routine is almost identical to and is accomplished by the ESC-Y sequence.

The ESC-Y sequence is four characters long. It is composed of an escape (decimal 27), a Y (decimal 89), the desired row plus 31 decimal (sum varies from 32 to 39), and finally the desired column plus 31 decimal (sum varies from 32 to 71). Building up this escape sequence takes many bytes of instructions. The call to POSIT at 427C is always more economical. To see this, look at the code at 427C-4289:

427C	3E	59		MVI A,59	;LD A,59 "Y"
427E	CD	70	42	CALL ESCA	
4281	7D			MOV A,L	;desired row
4282	D6	1F		ADI 1F	;make it printable
4284	E7			RST4	;print it
4285	7C			MOV A,H	;desired column
4286	C6	1F		ADI 1F	
4288	E7			RST 4	
4289	C9			RET	

The routine is interesting for several reasons. First of all, it is the only four character escape sequence for the LCD. Second, it illustrates that characters in the escape sequence must be greater than 32 decimal, (i.e. printable) so that the RST 4 routine won't mishandle them. Third, it shows what lengths Microsoft went to to make sure that the ROM operating system is cleanly structured. Virtually all routines affecting the screen are, at bottom, communicated to the screen through the RST 4 routine. This allows the programmer in charge of RST 4 to be sure that he has exclusive control over the inner workings of the screen.

POSIT is handy for moving the cursor about, as well as for returning it to its former position when printing. To do this, get the current cursor position with LHL D F639, store it with PUSH HL, print at the screen as desired, and then execute POP HL and CALL 427C.

### Unpublished ROM Routines for the LCD

Several routines are available which relate to the LCD, other than those published by Radio Shack. These routines may change if the ROM is altered creating problems.

A routine at 001E simply sends a space to the screen. A call to this location would take up three bytes, which is no savings over simply loading a 20 hex to the accumulator (two bytes) and calling RST4. This routine would only save bytes if you were at the end of a subroutine and needed to print a space, then return. A jump (not a call) to 001E could do both at once.

Another nice routine is located at 11A2. Assume HL points to a string of values (known to be printable) terminated with a 00 hex. Calling 11A2 will send that string to the screen. An identical routine is located at 5A58.

Finally, a routine at 1BE0 prints up to 256 characters to the screen, filtering out unprintable characters. The values are pointed to by HL, and the number of values to print is stored in the B register. (To print 256 values, load 0 in the B register.) This routine is handy for memory dumps. One drawback is that carriage returns and line feeds are suppressed. Recall that with this routine, as with any routine using RST4, the output may be routed to the printer simply by changing the value at F675.

The routine at 27B1, in a rather circuitous way, sends to the screen the character string pointed to by HL and ending with either an 0 or a quotation mark.

The routine at 5791 sends to the screen the character string which is pointed to by HL and ends with either a 0 or a quotation mark. The whole output is preceded by a carriage return if the cursor is not already located at the left edge of the screen.

11

# 14

---

## The Bar Code Reader

---

The Model 100's 9-pin connector labeled BCR with hardware designation CN2, can be used to attach a bar code reader wand. Pins 5 and 7 are grounded, pin 9 is a 5 volt source, and pin 2 provides data to the Model 100 from whatever is plugged into the BCR connector. The pins are shown in the illustration on page 209 of the Model 100 user's manual.

The signal at pin 2 has two hardware designations— it is called RXDB on page 209 of the user's manual and RXD on page 4-11 of the service manual. Regardless of the designation, it goes two places in the Model 100 — to one of the interrupt pins of the CPU and to bit 3 of input port BB.

The bar code reader interface circuitry is shown in figure 14.1. The power for the wand comes from pin 9 of CN2 and is designated VDD. The phototransistor signal enters the Model 100 at pin 2, designated RXD. After inversion, the BCR signal goes to bit 3 input port BB, implemented in hardware by port C of the PIO chip. The signal path to the CPU interrupt pin is designated RST 5.5.

The following BASIC program affords an easy means of examining how the input port functions:

```
1 IF 8 AND INP(187) THEN BEEP:GOTO 1 ELSE 1
```

Due to the pull-up resistor R70 at the input terminal, the expression `8 AND INP(187)` evaluates to zero when nothing is attached to CN2. As a result the ELSE clause of the IF statement is executed, and no beep is emitted.

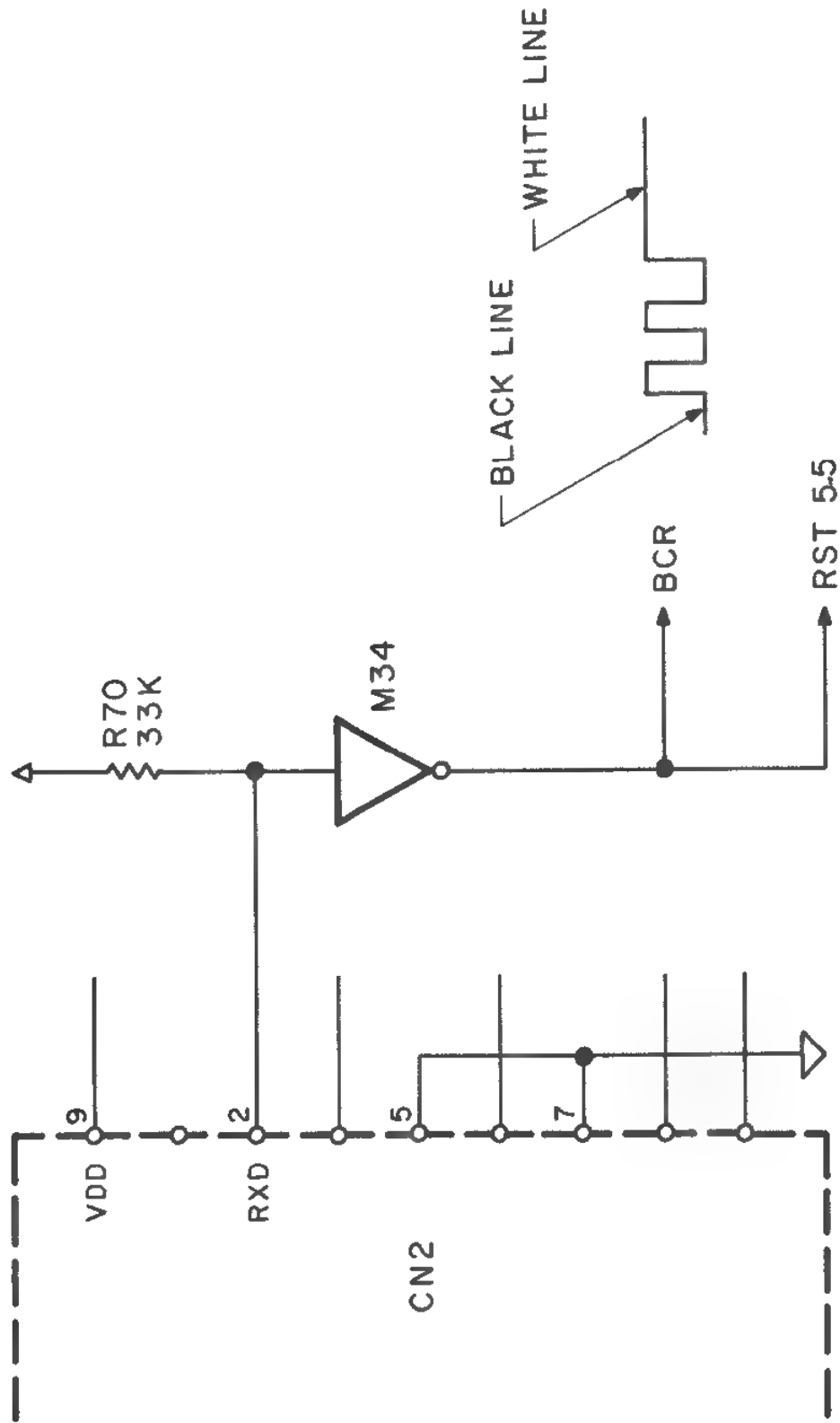
With a BCR wand attached (and no custom BCR software loaded), touching the wand to a white surface should produce a beep. This is because the wand contains a light source, powered by pin 9, and a phototransistor, which grounds pin 2 when light is reflected back to it.

The interrupt capability of the BCR interface works through what is called the RST 5.5 input of the 8085. Grounding CN2 pin 2 sends an interrupt signal to the CPU. The interrupt status of the CPU, which has been previously set by the SIM and EI or DI instructions, determines whether an interrupt will actually occur in response to the interrupt signal.

## **Determining When to Start Reading a Bar Code**

The most elegant (and complicated) method of determining when reading of a bar code has commenced is through the hardware interrupt. The EI and DI instructions enable and disable interrupts (*see* chapter 15). Thus, the DI instruction prevents the occurrence of an interrupt due to grounding of CN2 pin 2.

If interrupts are enabled, namely if the EI instruction has been executed more recently than the DI instruction, then the most recent update of the interrupt mask (done through the SIM instruction) determines whether the BCR signal will be able to cause an interrupt.



**Figure 14.1.** Bar code reader interface circuitry

The SIM instruction (*see* chapter 2) fulfills either of two functions depending on the contents of the accumulator. If bit 3 is set, the interrupt mask is updated, based on bits 0, 1, and 2. If bit 6 is set, the cassette output pin is updated, based on bit 7. It is possible to update both, by setting bits 3 and 6. However such a practice is generally not undertaken.

Of the three interrupts controlled by the interrupt mask, it is bit 0 which controls the bar code reader interrupt. Setting that bit masks, or defers, any BCR interrupts. The present Model 100 ROM sets that bit whenever it updates the interrupt mask, so that BCR interrupts are never enabled.

Since the present contents of the interrupt mask are available to the CPU through the RIM instruction (*see* chapter 15), it is possible to read the interrupt mask into the accumulator, set or reset bit 0, set bit 3, and execute the SIM instruction. This action masks or unmasks the BCR interrupt without disturbing the status of the interrupt masking for the other two interrupts (UART data ready and clock/calendar timing pulse).

## **Handling the Interrupt**

As discussed in chapter 15, when the BCR interrupt is enabled and unmasked and when the interrupt occurs due to the wand encountering a white surface, a subroutine call to 2CH occurs. 2CH is located in ROM; this location is a jump to F5F9. The user should put a jump to the interrupt-handling routine at that address. The routine should end with an EI and a RET.

## **Polling the Bar Code Reader**

BCR data can be input without using interrupts using a technique called *polling*. When BCR data is to be input, the CPU repeatedly inspects the incoming signal at bit 3 of input port BB. Usually the signal is a logic zero, so a transition to a logic one indicates scanning has begun. Since the remainder of the bar code is read rather quickly, the CPU must monitor the input port very closely. The CPU ignores any other inputs except the break key.

## Reading the Bar Code

Reading the bar code is a complicated matter. Assuming the wand moves along the image at approximately 33 inches per second, a code one inch wide has come and gone in 30 milliseconds of read time. There may be thirty bars in the code, meaning that sixty black/white or white/black transitions must be detected, with an average of about 500 microseconds in between. The time interval between detection of each pair of transitions must be noted for later analysis.

Here is a routine for detecting a pair of transitions, assuming a white region has just been detected:

```

LOOP:      MVI D,00          ; how long is white band?
           IN BB             ; get BCR data
           ANI 08            ; get bit 3
           INR D
           JNZ LOOP          ; if still white, loop
           IN BB             ; it's black, look again
           ANI 08            ; maybe one black value
           INR D
           JNZ LOOP          ; was a fluke
           MOV M,D           ; (HL) contains width of bar
           INX HL
           MVI D,00          ; how long is black band?
LOOP1:     IN BB             ; get BCR data
           ANI 08            ; get bit 3
           INR D
           JZ LOOP1          ; if still black, loop
           IN BB             ; it's white, look again
           ANI 08            ; maybe one white value
           INR D
           JZ LOOP1          ; was a fluke
           MOV M,D           ; (HL) contains width of bar
           INX HL

```

The loops in the above routine each last 31 machine cycles, or about 13 microseconds. The D register averages a final value of 40; less if the band was narrow and more if it was wide.

When the entire code has been read, the intervals must be analyzed to see which bars and gaps were wide and which were narrow. Then, based on the code being used (Universal Product Code, 3 of 9,



Interleaved 2 of 5, Codabar, etc.), the wide/narrow information must be translated into digits or ASCII characters.

If the coding system includes a parity check, it is a good practice to announce bad parity with a distinctive tone, giving the user the opportunity to scan the pattern again.

### **Using the BCR Connector for Purposes Other Than Reading Bar Codes**

Devices other than a wand can be plugged into the BCR connector. Any information source represented by a short, or open, between pins 2 and 5 of CN2 may be read by the CPU as discussed previously.

It is a good practice to use an optocoupler to protect the BCR interface. Use a standard coupler such as Radio Shack 276-1654, wired as shown in figure 14.2. The resistor limits current to twenty milliamperes, protecting the five-volt supply and the light-emitting diode at pins 1 and 2 of the coupler. Closing the switch causes a logic one at bit 3 of input port BB.

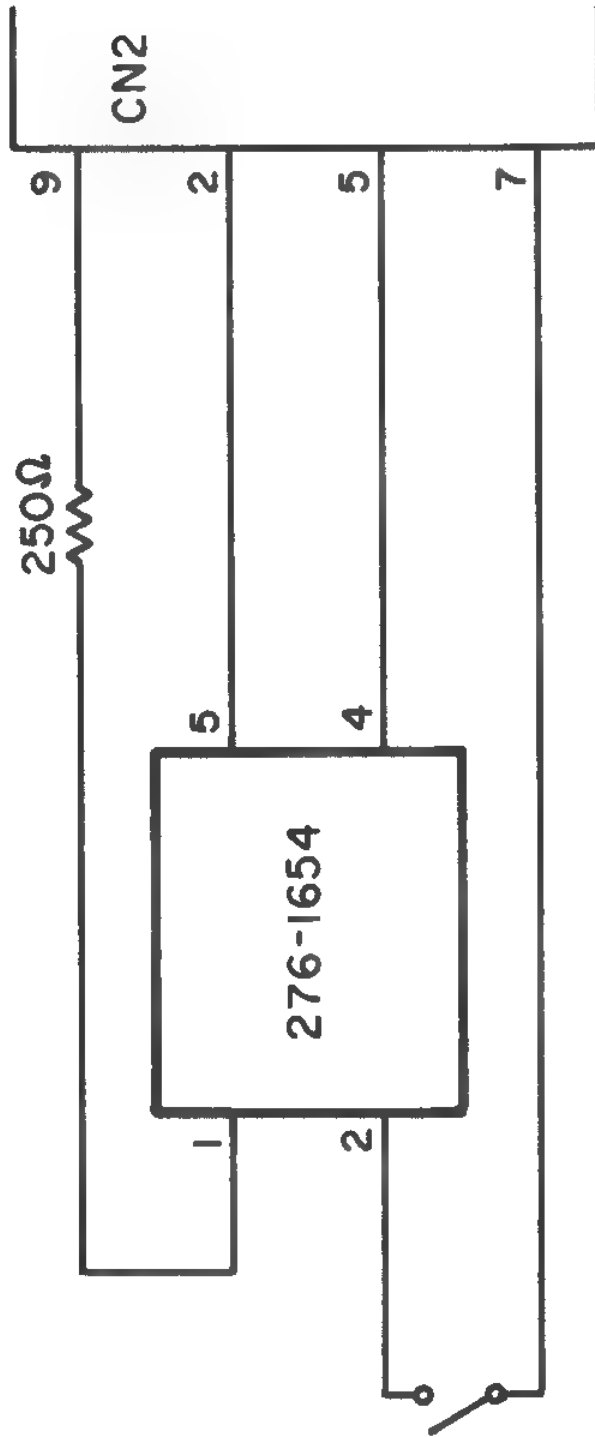


Figure 14.2. Discrete input at BCR connection



# 15

---

## Interrupts

---

Rarely does any event proceed from start to finish without being interrupted. For example, if you are reading a book and the telephone rings, you are likely to put a bookmark in the book, answer the phone, and later return to your reading.

Often in programming a computer to accomplish a task, the equivalent of “answering the phone” is a useful addition to that program. For example, when the Model 100 is executing TELCOM, it monitors the keyboard to see if a key had been pressed since its last check. When a key is pressed, the Model 100 must respond accordingly. These responses might include: opening a file for uploading or downloading, scrolling the screen, or transmitting a character to the distant computer by way of the UART.

For these reasons, and for many others that will be apparent, the Model 100 uses *interrupts* for several time-related functions. Interrupt can be defined as a means by which program execution is interrupted by a stimulus which originates outside the CPU. The CPU's usual sequential execution of instructions is brought to a temporary halt, and control is transferred to program instructions stored at an interrupt address associated with that particular interrupt. The CPU address being executed at the time of the interrupt is stored on the stack. This allows CPU instruction execution to resume properly once the interrupt instructions have been executed.

When the conditions prompting the interrupt has been satisfied, a properly written program invokes the RET instruction, returning program control to the address about to be executed prior to the arrival of the interrupt signal.

To accomplish all this, the Model 100 incorporates:

- a CPU designed to be able to respond to several interrupt signals
- a hardware layout connecting several interrupting devices to the CPU
- a ROM operating system which provides routines for the CPU to perform in the event of an interrupt condition.

The 8085 can accept as many as five separate electrical interrupt signals from other devices. In that respect it differs from its closest relatives, the 8080 and the Z80, which each accept only one.

The five interrupt signals to which the 8085 will respond are shown in table 15.1. This table also depicts the address to which the 8085 transfers control and the interrupt source to which the 8085 is connected in the Model 100.

Pin 6 of the 8085, the TRAP line, is wired to the power supply (*see* figure 16.1). A signal designated LPS is generated when the battery or external power source drops below 3.7 volts. *See* chapter 16 for a discussion of the power supply.

This is not the same as the low battery warning light, which illuminates much sooner than the LPS signal, namely when the power

drops below 4.1 volts. The CPU cannot detect the low battery warning light.

The internal design of the 8085 is such that the LPS signal causes what is functionally equivalent to a subroutine call to 24H. Disassembly of the ROM reveals a jump to F602 at that location. F602 generally contains a jump to 1431, which in turn results in:

- storage of a few crucial parameters that are used later upon power-up.
- a powering down of the Model 100 through bit 4 of output port 178 or 186, the power-control signal

You could store a different jump at F602 if a different means of handling low power was desired. Such a handler must end with a jump to 1431.

Pin 7 of the 8085, the RST 7.5 line, is wired to the clock/calendar chip. A signal designated TP is generated approximately once every four milliseconds. The actual pulse rate is 256 hertz. The 8085 examines 3CH to determine the next action. The ROM instruction there is DI (disable interrupts) followed by a jump to 1B32, which is a call to F5FF. That part of RAM usually contains a RET (return), though you could store a jump there to a routine to be prompted by the TP signal. Such a routine must end with a RET instruction.

The return brings control back to the TP-interrupt routine at 1B35, which performs a variety of housekeeping activities and ultimately returns control to the interrupted program.

Note it is possible to reprogram the clock/calendar chip to generate the TP signal at rates other than 256 hertz. See chapter 11.

Pin 8 of the 8085, the RST 6.5 line, is wired to the Universal Asynchronous Receiver/Transmitter (UART). A signal designated DR (Data Ready) is generated when the UART has received a character, whether from the RS-232C port or from the phone modem. The RST 6.5 interrupt generates a subroutine call to 34H, which contains a DI opcode and a jump to 6DAC, which in turn calls F5FC. Usually F5FC contains a simple return; the routine at 6DAF then retrieves the received character and places it in a buffer. You could put a subroutine call or jump at F5FC to handle incoming characters differently.

Pin 9 of the 8085, the RST 5.5 line, is wired to the BCR socket.

The internal design of the 8085 results in what is functionally equivalent to a subroutine call to 2CH. Disassembly of the ROM at

that location reveals a DI instruction followed by a jump to F5F9.

F5F9 generally contains a simple return, assuming the bar-code reader driver has not been loaded. The user could put a jump instruction at F5F9 if different handling of a received BCR signal were desired; such a handler must again end with a return.

Pin 10 of the 8085, the INTR line, is wired to the Expansion Bus connector, pin 17. A signal designated INTR (Interrupt) is generated when that pin is pulled high by a device plugged into that connector.

The response of the 8085 to this interrupt is identical to that of the 8080 and similar to that of the Z80. A Restart subroutine call occurs to the location jammed onto the address bus by the interrupting device. A detailed discussion of interrupt jamming is beyond the scope of this book. For further information, *see* Larsen, Titus and Titus, *8080/8085 Software Design Book 2*, chapters 2 and 3 and Leventhal, *8085 Assembly Language Programming* chapter 12.

### **Interrupt Priorities**

The 8085 is designed with a priority circuit that decides which of two simultaneous interrupts will determine the interrupt routine to be followed. The priorities are detailed in table 15.2.

### **Masking and Disabling of Interrupts**

Handling of an interrupt, even if it simply results in a RET instruction, takes time. Since this interval can throw off time-sensitive functions, such as cassette I/O. It is necessary to be able to disable, or mask, most interrupts. This is accomplished through the EI, DI, RIM, and SIM instructions.

As is indicated in table 15.2, the LPS interrupt cannot be masked, although the other four can be. You can either disable all four through the DI instruction or enable interrupts through the EI instruction.

In the 8080 and Z80, this reflects the full range of interrupt mask choices. In the 8085, however, three of the four interrupts (TP, DR, and BCR) may be individually masked, through the use of the SIM instruction, *see* table 15.3.

Masking accomplished by the SIM instruction is cumulative with that accomplished by the DI instruction. Enabling of, say, the DR interrupt requires both an EI and the appropriate SIM.

As a result, certain combinations of enabled and disabled interrupts are not possible. For example, no set of instructions can bring

about an enabled DR interrupt and at the same time a disabled INTR interrupt.

Let's look at how the ROM routines enable and disable the various interrupts. Disassembly of the subroutine at 765C reveals:

765C	F3	DI	
765D	3E 1D	MVI A,1D	(LD A,1D)
765F	30	SIM	
7660	FB	EI	

At 765D, the accumulator is loaded with 1D, which is 00011101 in binary. Bits 4, 3, 2, and 0 are on. The next instruction, SIM, loads the contents of the accumulator into the interrupt mask. It is important that bit 6 be off, so that this execution of the SIM instruction will leave the cassette output (SOD) unchanged. Because bit 5 is on, the TP interrupt, if pending, is reset.

Because bit 3 is on, the mask values in bits 2, 1, and 0 are updated. The result is that the DR interrupt is enabled, but the TP and BCR interrupts are disabled.

Strictly speaking, it is not the SIM instruction that enables the DR interrupt; SIM merely sets the stage. It is the EI in the following line that enables the DR and the INTR interrupts.

Other ROM routines enable different combinations of interrupts. For example, the routines at 1B3B and 457D enable the DR interrupt but not the TP or BCR interrupts. They do not reset the TP request. The routine at 6CE4 resets the TP request and enables the DR and TP interrupts but not the BCR interrupt. The routines at 4584, 6D69, 71F6, 726D, 73EA, and 743E enable the DR and TP interrupts but not the BCR interrupt.

When an interrupt is handled by the 8085, all other interrupts are disabled, just as if a DI instruction had been executed. As a result, no other interrupt, whether higher or lower in priority, is handled. You should reen able interrupts somewhere in the interrupt-handling routine. This is usually done just before the final RET instruction. The actual reenabling occurs after the completion of execution of the instruction immediately following the EI. If the 8085 had been designed so that reenabling occurred with the EI instruction itself, the



presence of any pending interrupts would take up too much space in the stack area.

Radio Shack Model I and III programmers know that disabling of interrupts using CMD "T" interfered with system timekeeping. No such problem exists within the Model 100, as time is kept by a clock/-calendar chip whose function is unaffected by CPU status and which continues to function after the Model 100 is turned off.

Masking of interrupts is used to great advantage in Model 100 BASIC. The commands ON COM GOSUB and ON MDM GOSUB are driven directly by the DR interrupt and are masked by COM STOP and MDM STOP.

Similarly, the commands ON KEY GOSUB and ON TIMES\$ GOSUB are driven by the TP interrupt and masked by KEY STOP and TIMES\$ STOP.

### **Uses for the RIM Instruction**

The RIM instruction is used for cassette input (*see* chapter 12). This is, in fact, the only purpose to which it is put by the Model 100 ROM. However, RIM can also be used to read the mask status of certain interrupts.

Suppose you desired to change only one bit of the interrupt mask — for instance to enable, the DR interrupt while leaving unchanged the masked or unmasked status of the TP and BCR interrupts. This could be accomplished by:

```
RIM
ANI 05      (AND A,05)      ; trim bits 1 and 3-7
ORI 08      (OR  A,08)      ; turn on bit 3
SIM
```

This routine's RIM instruction reads the present mask status of the TP and BCR interrupts (bits 0 and 2 of the interrupt mask). It then turns off bits 1 and 6 (through the AND instruction) to enable the DR interrupt. To leave the SOD signal undisturbed, it turns on bit 3 to allow updating of the interrupt mask and updates the mask with the SIM instruction.

Suppose you wished to poll the UART DR line, rather than service it through the interrupt routine at F5FC. To 8080 and Z80

programmers this would appear to be impossible in the Model 100 as a hardware matter, since the UART DR line does not go to any I/O port circuitry (such as the PIO) but instead goes only to CPU pin 8, which is an interrupt pin.

The 8085 RIM instruction, however, allows such polling. As shown in table 15.4, after a RIM, bits 4, 5, and 6 of the accumulator indicate whether BCR, DR, and TP interrupts, respectively, are pending.

**Table 15.1.** Types of Model 100 interrupts

CPU Pin	8085 Designation	M 100 Designation	Jump Address	Source
6	TRAP	LPS	24H	Power supply
7	RST 7.5	TP	3CH	Clock/calendar chip
8	RST 6.5	DR	34H	UART
9	RST 5.5	BCR	2CH	Bar-code reader
10	INTR	INTR	varies	Expansion bus

**Table 15.2.** Interrupt priority and masking

Priority	Signal	How Masked
1	Low power signal	Nonmaskable
2	Timing pulse	DI or SIM bit 2
3	Data ready	DI or SIM bit 1
4	Bar code reader	DI or SIM bit 0
5	Expansion bus	DI

**Table 15.3.** Set interrupt mask (SIM) configuration

Bit	8085 Function	Model 100 Use
0	RST 5.5 mask	Ignore BCR interrupts
1	RST 6.5 mask	Ignore UART DR interrupts
2	RST 7.5 mask	Ignore 256-Hz interrupts
3	mask set enable	Mask set enable
4	reset 7.5 interrupt	Reset 256-Hz interrupt
5	not used	Not used
6	SOD set enable	Allow updating of cassette output flip-flop
7	SOD	Data for cassette output

**Table 15.4.** Read interrupt mask (RIM) configuration

Bit	8085 Function	Model 100 Use
0	RST 5.5 enabled	BCR interrupt enabled
1	RST 6.5 enabled	UART DR interrupt enabled
2	RST 7.5 enabled	256-Hz interrupt enabled
3	IE status	IE status
4	RST 5.5 pending	BCR interrupt pending
5	RST 6.5 pending	UART data ready
6	RST 7.5 pending	256-Hz pulse pending
7	SID	Cassette data in

# 16

---

## The Power Supply

---

The Model 100 is designed to draw its power from four AA cells or from an AC adapter (catalog number 26-3804). Main power is controlled by the ON/OFF switch SW-5 on the right side of the unit.

With no peripherals attached and no sound being emitted from the beeper, the Model 100 draws about 350 milliwatts. Radio Shack maintains that under full load conditions it may draw as much as 1100 milliwatts, although this author has never seen it draw more than about 975 milliwatts.

The actual current drain at a given instant is a function not only of the internal load but also of the voltage supplied. This is because the Model 100 contains a DC-to-DC convertor which creates all needed voltages from whatever DC level is supplied. The convertor requires a

certain amount of source power (source voltage multiplied by source current), so that if the DC source is of a lower voltage, the convertor makes up for it by drawing more current. It is able to do this with input voltages ranging from 7 volts (from the AC adapter) to as little as 3.7 volts (from batteries that have run down almost to the point of prompting a CPU-induced power-down).

## **The DC-to-DC Convertor**

The most complicated part of the Model 100 power supply is the DC-DC convertor, shown in the upper right corner of figure 16.1.

Transformer OT2, on the right, is used to provide the various voltages. The designers of the Model 100 took advantage of the fact that transformers are more efficient at higher frequencies; transistors T21 and T22 provide an alternating current of 100 kilohertz to the primary winding. This is substantially higher than the 60 hertz alternating current used in the transformer of the AC adapter.

The currents from the center-tapped secondary winding are rectified, filtered, and regulated to produce the various voltages required in the computer. Minus 5 volts, designated VEE, is provided for the RS-232C driver, for the various operational amplifiers in the cassette and modem circuits, and for the liquid crystal display. Plus 5 volts, designated VB and backed up by a nickel-cadmium (nicad) cell, is presented for the RAM memory and clock/calendar chip. Plus 5 volts, designated VDD, is provided for everything else in the computer.

## **Memory Power**

Memory protection is provided by a nicad cell rated 3.6 volts at 50 milliamperes-hours. The cell, shown in figure 16.2, appears in the schematic as 3-51FT and bears reference number P-36 in the service manual. Since the rated protection time for a 32K machine is eight days, this means that the 32K of RAM together with the clock/calendar chip draw something less than 0.25 microamperes.

If a nicad is repeatedly discharged only halfway and then recharged, it will lose the second half of its capacity. Thus, most nicads should always be discharged fully before recharging. Nicads are known as *discharge memory* devices due to their propensity to lose the unused portion of their capacity when they are not discharged completely before recharging.

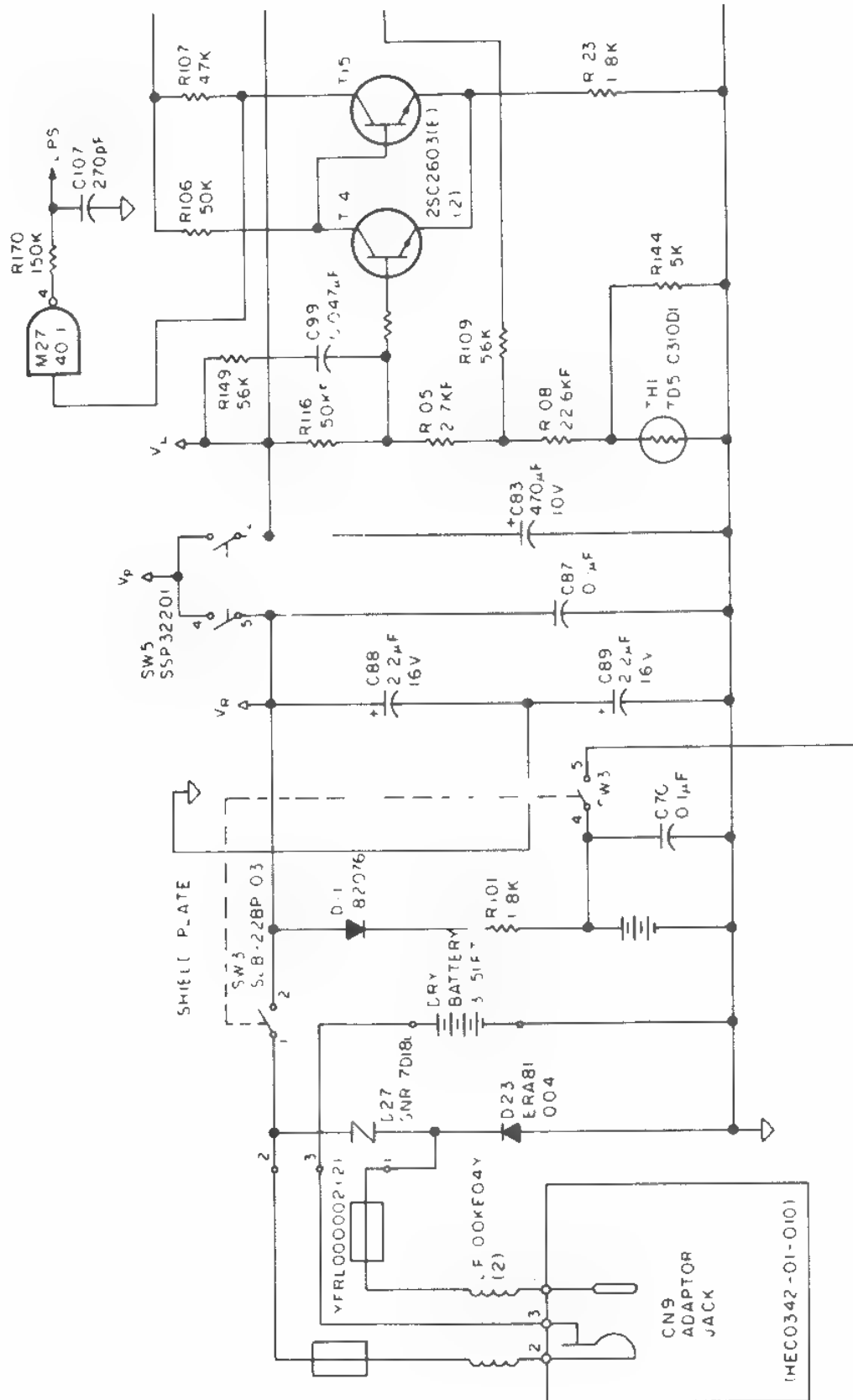


Figure 16.1. Power supply and reset circuit

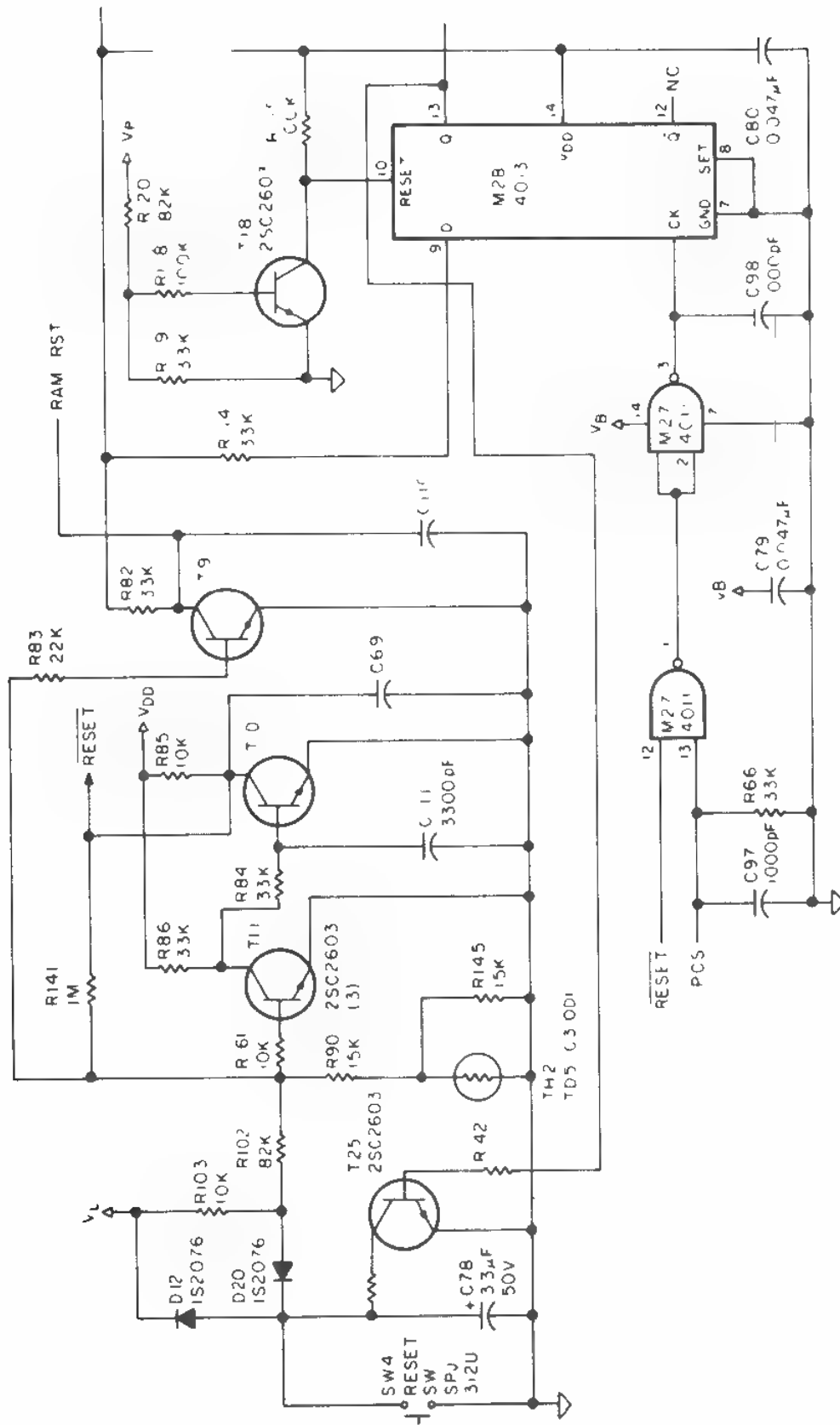
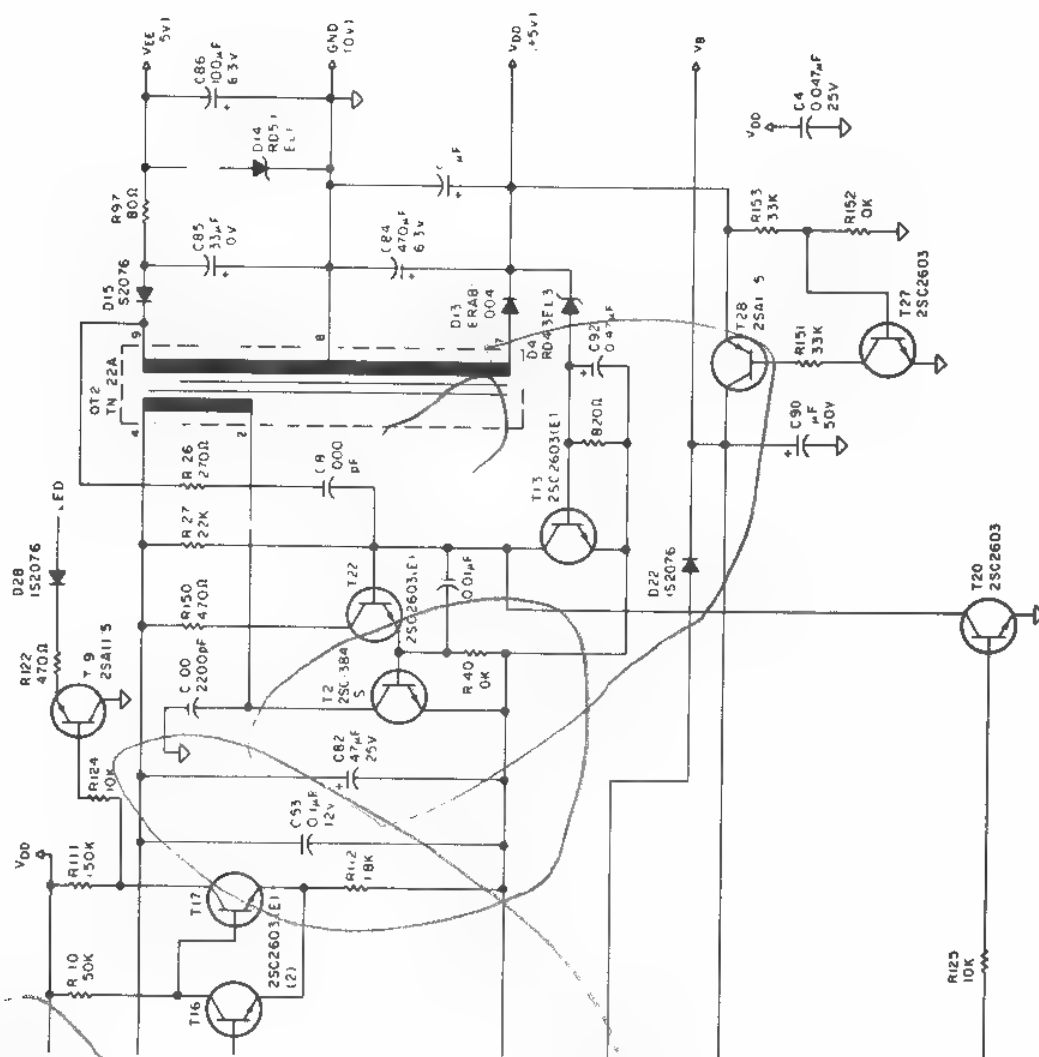


Figure 16.1.(cont.) Power supply and reset circuit



**Figure 16.1.(cont.)** Power supply and reset circuit



In the Model 100, however, the nicad is recharging whenever the four AA cells are in place and the AC adapter is connected. The only time the nicad discharges is during the interval when the "low battery" light comes on and when fresh batteries are inserted. Therefore, always keep a spare set of AA batteries on hand to be installed as soon as the LOW BATTERY light comes on.

Before servicing the computer and during installation of expansion RAM modules, you must remove all sources of power from the circuit board. Because the nicad is soldered in place and cannot easily be removed, a switch SW3 is provided to disconnect the nicad. This double-pole single-throw slide switch labeled "memory power" is located on the bottom of the computer. Turning it off for more than a few seconds results in loss of all data in RAM and loss of proper clock/calendar timekeeping.

It is hard to imagine any reason to turn off memory protection other than for servicing the computer. If the computer is going to be stored for a considerable length of time, it is a good practice to remove the AA cells as a precaution against leakage. In any event you should back up all important files on tape, since you cannot rely on the nicad lasting more than a few days.

An extended storage period provides the ideal opportunity to discharge the nicad fully so as to restore its full capacity. To do so, leave the memory power switch on.

## **Low-Power Signals**

The power supply includes circuitry to warn both you and the CPU of impending battery loss. A voltage divider, located at the upper center of figure 16.1 and composed of TH1, R144, R108, R105, and R116, yields voltages calculated to trigger the "low battery" light-emitting diode through transistors T16 and T17. The voltage divider also triggers LPS, the low power signal, through T14 and T15.

The connection between the "low battery" transistors and the LED is rather circuitous. The transistors drive T19, which is wired to CN8 on the main printed circuit board. A two-wire cable is plugged into CN8 whose other end is soldered to location CN3 on the LCD panel, which is in turn wired to the LED itself, designated P-24.

The divider is set up so that as the supply voltage falls, the LED illuminates first (at about 4.1 volts). Later LPS is activated (when the supply has fallen to about 3.7 volts). There is no connection from the LED circuit to the CPU. The LPS signal is the first clue the CPU has that the supply voltage is falling.

Since the switching thresholds of the transistors are affected by temperature, thermistor TH1 is provided to improve consistency of circuit response over a range of temperatures.

Whenever power drops far enough to activate the LPS transistors (and this includes simply turning off the main power switch SW-5), LPS goes to two places. It appears at bit 7 of input port D8 as a logic zero (changed from the usual logic one) and also activates the TRAP interrupt line of the 8085 CPU.

The TRAP interrupt, which cannot be masked or disabled, disables all other interrupts and causes a subroutine call to 24H. This results in an orderly termination of calculations in progress. The computer then turns itself off by turning on bit 4 of output port BA. The resulting signal, called PCS (power control signal) toggles flip-flop M28, which appears at the bottom of figure 16.1. An output of the flip-flop inhibits the feedback which usually sustains the convertor oscillator, cutting off energy to the VEE and VDD lines. As a result, all activity stops except clock/calendar timekeeping and RAM data preservation. The ROM code for the power down is located at 1431 through 1458.

## Reset Circuitry

When power is turned on, circuitry at the lower left of figure 16.1 provides a RESET-NOT signal to initialize the CPU, the PCS circuitry, the LCD, and flip-flop M14. The CPU provides RESET\* to initialize the modem, the UART, the PIO, flip-flop M36, and any device plugged into the Expansion Bus. The reset circuitry also provides a RAM RST signal, which disables the RAM chips to protect their contents during power-down. This signal is also sent to the Expansion Bus to protect any RAM that is installed there. The single-pole single-throw RESET button SW-4 at the rear of the computer provides the same reset signals as does the on/off switch.

## Powering Up The CPU

The 8085 CPU begins program execution at memory location 0000. The ROM code at that location jumps to 7D33, where, after a 100-millisecond delay, the PIO is initialized. (See chapter 5 for a discussion of the PIO.) If CTRL-BREAK was pressed during the reset or power-on, if the amount of installed RAM has changed, or if the RAM file directory entry for BASIC at F5F0 is missing, a "cold start" is performed complete with a purging of all RAM files.

Assuming the RAM directory has been found to be in order and no new RAM has been installed, the routine checks to see if it can reload the pointers that were in effect when the power-down occurred. The stack pointer, for instance, is retrieved from FABE.

## The AC Adapter

The AC adapter, catalog number 26-3804, is not UL approved. However, it is similar in design and construction to many UL-approved adapters, so there is no safety risk. Inside the adapter is a transformer, rectifier, and capacitor, as shown in figure 16.2.

The transformer has a 120-volt primary winding and a nominal 5.6 volt secondary winding rated at 400 milliamperes. The alternating current output is connected to a one-piece full-wave rectifier, which produces direct current, but with a substantial AC ripple. The DC current is smoothed by a 2200-microfarad, 10-volt electrolytic capacitor and provided to the Model 100 by a two-meter cable. The white-striped lead of the cable is the negative wire, which connects to the inner conductor of the round plug.

The plug is a 5.5-millimeter barrel plug, equivalent to Radio Shack catalog number 274-1551. The plug connects with the DC6V jack on the right side of the computer, internally designated CN9. Ferrite beads are installed on the internal wiring to CN9 to aid in suppression of radio frequency energy which might otherwise be transmitted into the house wiring through the AC adapter or into the air by the adapter cord acting as an antenna. Though the adapter is rated 6 volts at 400 milliamperes, the actual no-load output is about 8 volts. This drops to about 7 volts under a typical Model 100 load of 50 milliamperes or to about 6.5 volts under a heavy load of 150 milliamperes.

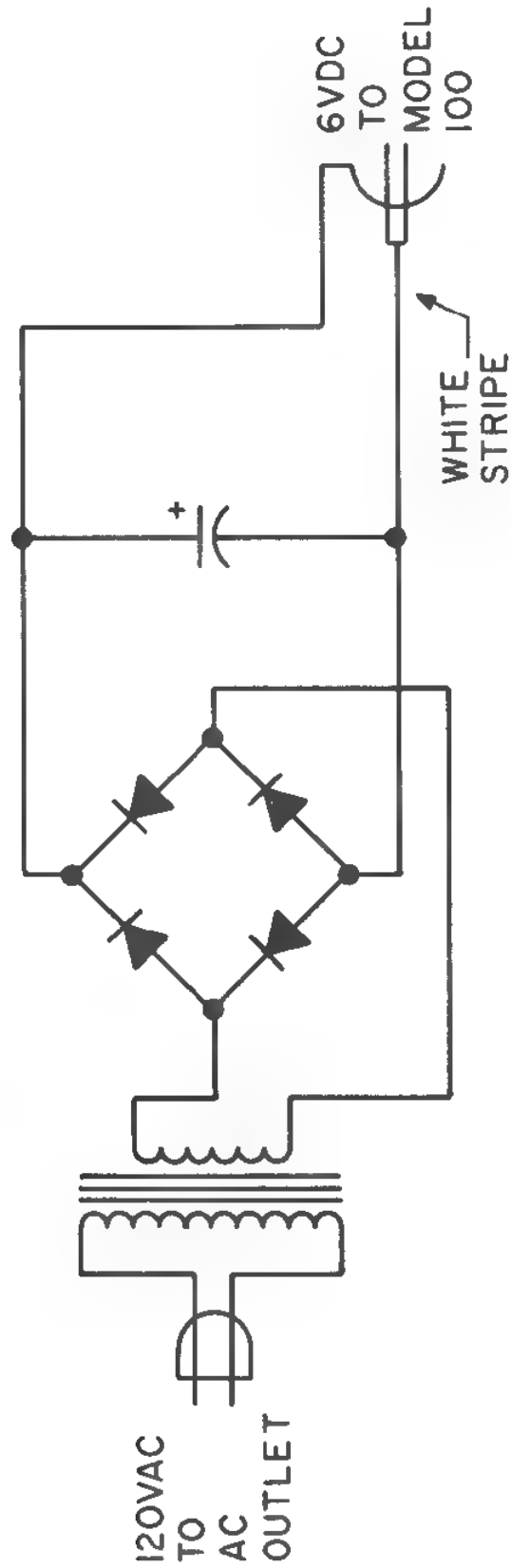


Figure 16.2. AC adapter

Since most AC adapters are composed simply of a transformer and rectifier without a capacitor, they do not perform any filtering. This explains why the Model 100 user's manual warns against using any adapter other than the Radio Shack adapter.

You could use a non-Radio Shack AC adapter which provides around 6 volts or so at 180 milliamperes, as long as capacitors are included which total 2200 microfarads. These may be spliced into the cord running from the adapter to the computer.

### **Alternative Power Supply**

In powering the Model 100, you are not limited simply to AA cells and the AC adapter. Any ripple-free source of DC between 3.8 and 7 volts and capable of providing at least 1.1 watts can be plugged into the DC6V jack using a barrel plug (Radio Shack catalog number 274-1551). Sensible choices include a six-volt lantern battery with screw terminals or three photocell arrays wired in parallel. Just remember that the negative lead must be wired to the center terminal of the barrel plug.

# 17

---

## Expansions

---

This chapter discusses a number of topics related to expansion of the Model 100. Hardware modifications or expansions would be required to implement each of these expansions.

### **The Bar-Code Reader and CRT**

The manner in which BASIC handles the opening of devices makes it clear that device specifications WAND and CRT will be available for the Model 100. When an OPEN is performed in BASIC, a file control block is set up with pointers to addresses that BASIC uses in performing output (PUT), input (GET), and file CLOSE, as well as any special handling required by that device. The routine addresses are stored in ROM in various locations and are listed in table 17.1.

Some devices cannot be used for certain directions of data flow. For example, input cannot be performed from the LCD, CRT, or LPT. Note that these table positions are empty.

In the case of the WAND and CRT device names, the ROM entries refer to RAM addresses that are shown here in table 17.1 in parentheses. For example, if a CRT OPEN is attempted, BASIC jumps to the address stored in FB1A.

The original ROM loads these RAM locations with a jump that results in a return to BASIC with a ?FC error. When expansion software is loaded, these locations in RAM are changed to jump addresses for the appropriate handlers.

**Table 17.1.** Subroutine addresses for BASIC device handling

Device	OPEN	CLOSE	PUT	GET	Special
LCD:	14D8	4D59	14E5		
CRT:	(FB1A)	4D59	(FB1E)		
CAS:	1689	16AD	16C7	16D2	1710
COM:	176D	179E	17A8	17B0	17CA
WAND:	(FB20)	(FB22)	?FC	(FB24)	(FB26)
LPT:	14D8	4D59	175A		55CD
MDM:	176C	17DB	17A8	17B0	17CA
RAM:	1506	158D	15AC	15C4	161B

The addresses in table 17.1 may be printed out by means of the BASIC program listed below.

```
5 OPEN"device"FOROUTPUTAS1
10 HX$="0123456789ABCDEF":AD=20721:FOR D= 0 TO 7:
GOSUB 1000: TA=PEEK(A)+256*PEEK(A+1):
FOR IT=TA TO TA+8 STEP 2:A=IT:GOSUB2000:
NEXT IT:PRINT#1,:NEXT D:CLOSE:END
1000 PRINT#1,CHR$(PEEK(AD));:AD=AD+1:
IFPEEK(AD)<128 THEN 1000 ELSE PRINT#1,"."+CHR$(9):
A=20755+(2*(255-PEEK(AD))): AD=AD+1:RETURN
2000 PRINT#1,CHR$(9):A=A+1:GOSUB 3000:A=A-1:GOSUB3000:RETURN
3000 PRINT#1,MID$(HX$,1+((PEEK(A)AND240)/16),1);
MID$(HX$,1+(PEEK(A)AND15),1):RETURN
```

## Unused Pins

Two pins in the Model 100 are available for hobby usage. The first is an input port signal at M23, pin 2. This pin, presently wired only to a pullup resistor, controls bit 6 of input port 208. If a switch were connected from this pin to ground, software could determine the position of the switch by ANDing the value at port 208 with 64. If the result is zero, then the switch is closed.

The other unused pin is M16, pin 14. M16 is the integrated circuit that controls I/O ports 128 to 255. Pin 14 is usually at 5 volts, but drops to 0 whenever I/O ports 144 to 159 are accessed by the CPU. This signal could be fed to other CMOS integrated circuits or, with suitable buffering, could be used to control devices outside of the Model 100.

## DISK INPUT/OUTPUT

The Model 100 designers have planned for disk or other bulk storage. The BASIC key word table contains DSKI\$ and DSKO\$; these words are handled at 5073 and 5071, respectively. Those ROM addresses contain references to RAM vectors at FB2E and FB30; which currently point to a ROM routine that yields a ?FC error. When DSKI\$ and DSKO\$ are implemented, these values change.

## ROM ROUTINES FOR BULK DATA TRANSFER

The routine at 7304 through 7326 makes reference to I/O ports 70 through 73, which are not presently implemented, but could easily be wired up at the expansion connector. This routine resembles a disk input routine, as it loads three numbers to output ports (drive number, track, and sector) and retrieves a specified number of bytes from an input port.

The routine at 767D through 770A appears to be a bulk output routine. It refers to ports 80, 81, 82, and 83, which are not presently included in hardware but which could easily be added at the expansion connector.

## RECORD I/O

It appears that some sort of record I/O is being planned for the Model 100. Two variables, LOF and LOC, are provided for. In most



BASIC versions, LOF carries the number of logical records in a file, while LOC carries the present logical record number within an open file, a portion of which has already been input.

The RAM vector for LOF is located at FB28, while the vector for LOC is located at FB2A.

### **LISTING FILES TO THE PRINTER**

The command LFILES is provided for, which you would expect to cause a listing of files at the printer. The RAM vector is at FB2C.

### **FUNCTION KEYS IN TELCOM TERM MODE**

The F6 and F7 function keys in Term mode presently do nothing. This is because the RAM vectors at FB0C and FB0E simply point to RETurns. If you want to put one to use, just place something else in the RAM vector location. As a demonstration, in BASIC type:

```
POKE 64268,41:POKE 64269,66
```

Whenever the F6 key is pressed, the computer jumps to the routine at the location  $41+256*66$ , which is the BEEP routine.

Many other RAM vectors exist. They are located from FADA to FB39 and are referred to by RST 7's. The byte following the RST 7 is used as an index into the table starting at FADA. Thus, RST 7 followed by 4E (which is what happens at the LOF routine at 506B) causes a reference to  $FADA+4E$ , or FB28. The address at FB28 and FB29 is jumped to by the RST 7 routine.

During a cold start, ROM loads the lower portion (up to FB12) of the FADA table with 7FF3, which is a RET. ROM loads the rest of the table (up to FB38) with 08DB, which was chosen because it returns to BASIC with an error 5, ?FC.

### **Understanding the Option ROM Socket**

Selection of the option ROM is accomplished by turning on bit 0 of output port E8, as shown in Figure 17.1. To do so properly, obtain the contents of E8, which are stored by ROM at FF45. OR with 01, and OUT the data to the port.

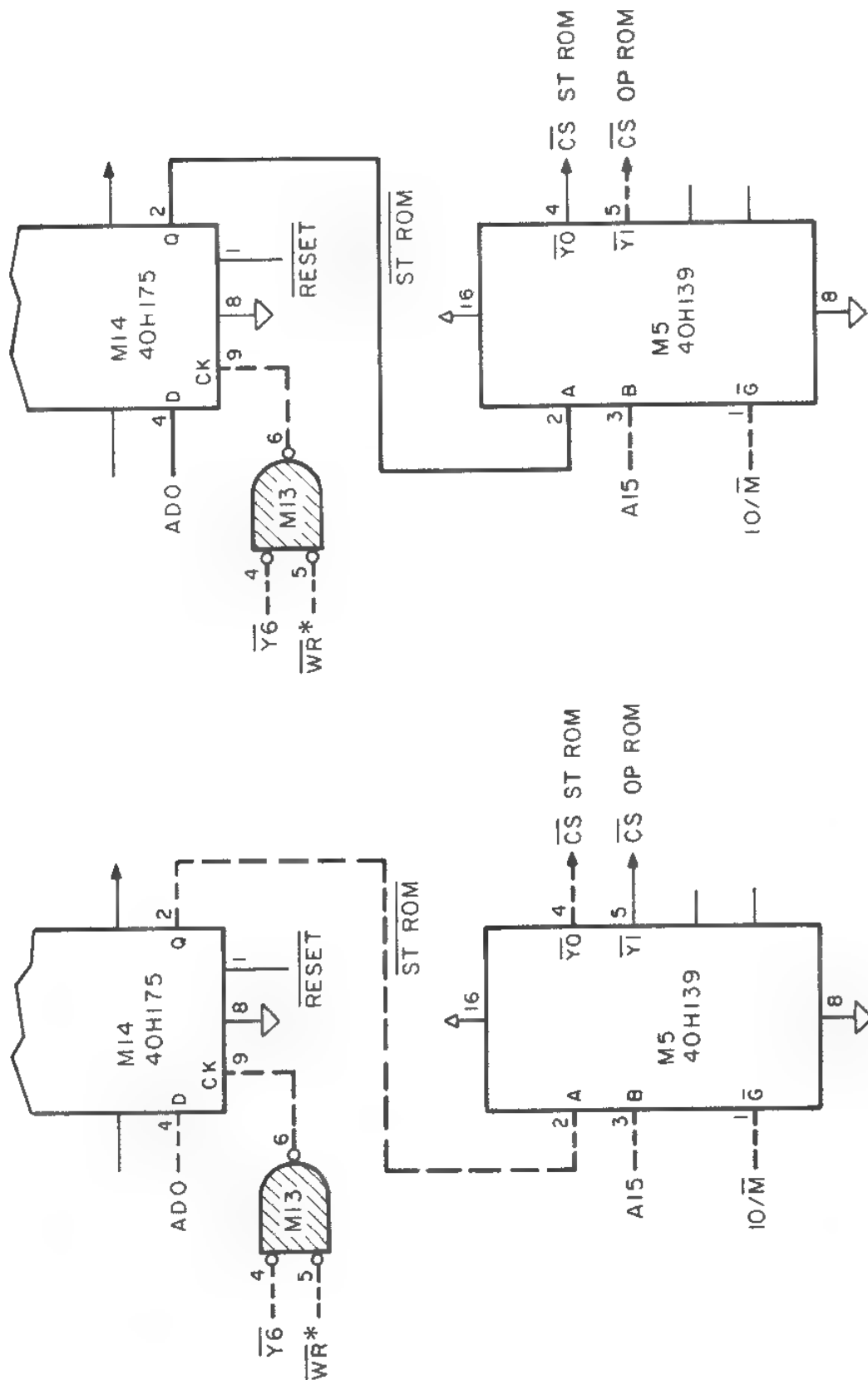


Figure 17.1. Option ROM selection

When a cold start takes place, the start routine checks the option ROM socket to determine the presence of a device (this occurs at 036F).

If location 40H in the option ROM contains 54 and location 41H contains 43, the ROM is assumed to be installed and functioning. A flag at F62A indicates whether the option ROM passed the test. The contents of ROM at 42H to 47H are treated as a filename. They are loaded into the RAM directory at F9BD through F9C2, immediately following SCHEDL. The filetype is specified to be F0 (valid entry, ASCII, machine language, ROM).

### EXPANSION BUS SOCKET

The expansion bus is made available in a socket on the underside of the unit. The socket is a standard forty-pin integrated circuit socket, so an IC header can be used to make connections. The service manual states that an *optional I/O control unit* and *RAM file unit* can be connected to the expansion socket. As discussed previously, there is room for growth in the RAM vectors for DSKI\$ and DSKO\$ and in the ROM routines to nonexistent I/O ports between 70 and 8F.

The signals at the expansion socket are listed in table 17.2 and are discussed in turn.

**Table 17.2.** Expansion socket signals

Pin	Name	Input or Output
1	VDD	output
2	GND	output
3	AD0	input/output
4	AD2	input/output
5	AD4	input/output
6	AD6	input/output
7	A8	output
8	A10	output
9	A12	output
10	A14	output
11	GND	output
12	RD*	output
13	IO/M*	output
14	ALE*	output
15	CLK	output

Continued on following page

16	A*	output
17	INTR	input
18	GND	output
19	RAM RST	output
20	NC	
21	NC	
22	NC	
23	GND	
24	INTA	output
25	RESET*	output
26	Y0*	output
27	S1	output
28	S0	output
29	WR*	output
30	GND	output
31	A15	output
32	A13	output
33	A11	output
34	A9	output
35	AD7	input/output
36	AD5	input/output
37	AD3	input/output
38	AD1	input/output
39	GND	output
40	VDD	output

All but a few of the signals at the expansion connector are derived directly from the CPU, with only the usual buffering.

### MEMORY ACCESS AT THE EXPANSION CONNECTOR

Since the Model 100 already contains devices which respond to every possible memory address, from 0000 to FFFF, the addition of a memory device via the expansion connector faces difficulties. As a result, unless you wish to cut traces on the printed circuit board, any addition of external memory could most easily be accomplished in port space rather than address space.

Nonetheless, memory addition is possible though difficult. Perhaps the easiest way would be to take advantage of the fact that the ROM from 0000 to 7FFF is already bank-switched. One could add circuitry putting 32K of RAM in 0000 to 7FFF whenever pin 27 (chip select not) of the option ROM socket was low. It would be necessary to copy into RAM those ROM routines that were needed for continued program execution.

It would appear the Model 100 designers have some sort of RAM expansion in mind since two reset signals are provided at the connector — the usual RESET\* signal (pin 25) and also the RAM RST signal (pin 19). The service manual states the RAM RST signal is to be used with external CMOS RAM.

### **Address Decoding**

All computers use signals like RD, WR, IO/M, address, and data for memory access. Memory address decoding in the Model 100 differs in one important respect, however, from that in most computers. Because the 8085 uses the same eight wires for data transfer and for address lines 0 through 7, any memory device added at the expansion connector requires address decoding circuitry which takes into account the ALE (address latch enable) signal present at pin 14. This signal determines whether the contents of the eight lines are to be interpreted by the other devices as address or data.

Other input/output status and control signals have been brought out to the connector. S0 and S1 are provided so that advance warning may be given to slow peripherals if a read or write is imminent. The signal designated A\* is a combined RD\* and WR\* signal.

### **Adding Ports to the Model 100**

Port space is a much more promising area of expansion than address space. For one thing, the port space is not yet filled; this is shown in figure 5.3. Also, because the port addresses are available on address lines 8 to 15, there is no need to worry about the ALE signal.

Finally, you can take advantage of circuitry in the Model 100 to decode port addresses. The port-select signal for ports 80 through 8F is already present at the connector — it is Y0\* at pin 26. In addition, the port-select signal for ports 90 through 9F is inside the Model 100 waiting to be used. It can be found at pin 14 of M16.

### **Parallel Port Input**

A typical general-purpose parallel input circuit is shown in figure 17.2. Using the configuration shown in the figure, the positions of the eight switches are available to the CPU at any input port in the range of 80 to 8F.

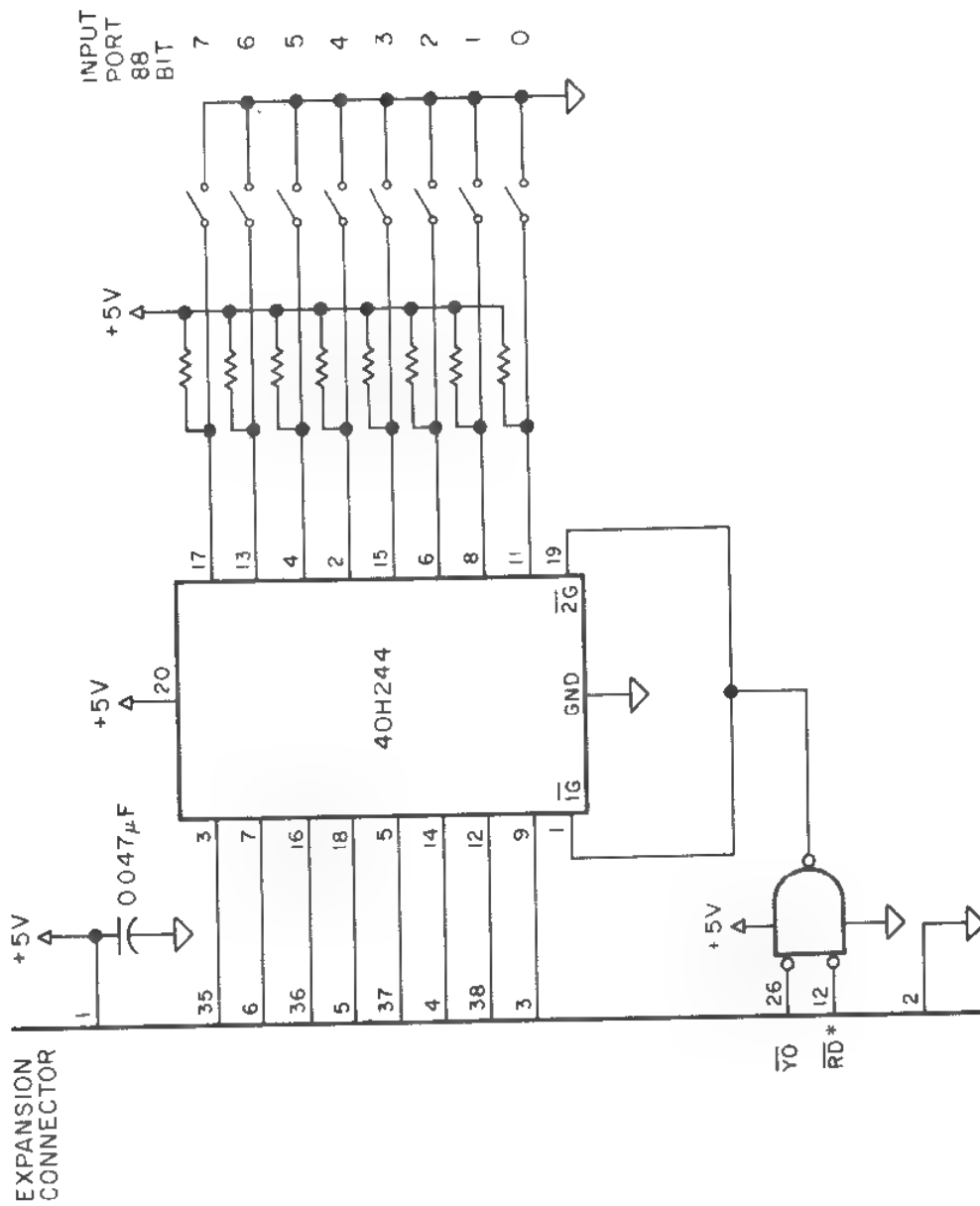


Figure 17.2. Parallel port input

## **Parallel Port Output**

Parallel output is most easily accomplished using flip-flops, as shown in figure 17.3. An output to any port in the range of 80 to 8F will load the byte in the accumulator into the flip-flops and will turn on the LEDs. A relay is shown which is turned on if the corresponding bit (bit 0) is a one.

## **Interrupts at the Connector**

The signals INTR and INTA, used for interrupt-driven programming, are available at the connector. If a device at the connector asserts INTR by pulling it up to +5 volts, the CPU responds to the interrupt. The response can be directed by jamming an interrupt vector address onto the address/data lines. This is not a technique for the idle experimenter. For more information, refer to the references cited in chapter 15 on interrupts.

### **THE TELEPHONE RING PULSE INPUT**

One intriguing circuit is that connected to pin 8 of the phone jack CN4, labeled ring pulse. This pin, if grounded, turns off bit 5 of input port D8. The easiest way to ground it is to short it to pin 2 of that connector. A simple hardware device, like that shown in Figure 17.4, can be connected to the direct-connect phone cable (catalog number 26-1410) and to the Model 100.

### **THEORY OF OPERATION**

When the phone line is quiet, direct current (DC) is present on the line without added ring-detect circuitry from the line by blocking the DC.

When the central office sends the ring signal, the capacitor couples the alternating current to diodes D1 and D2 which charges capacitor C1 to about 200 volts. This voltage, limited by resistor R2, allows a flow of about 20 milliamperes through the LED of optoisolator IC1, which turns on its phototransistor. Thus, during the ring pulse, Model 100 phone jack pin 8 is grounded through IC1 to phone jack pin 2.





The central office ring pulses occur about every six seconds and last about a second. When each pulse has ceased, capacitor C1 discharges quickly through the LED, turning off the LED and allowing the voltage at phone jack pin 8 to float back to its high level.

A parts list for a ring/detect circuit is provided in table 17.3. Any construction technique can be used. Lead lengths and placement are not critical, except that the wiring of pins 5 and 6 of the optoisolator should be carefully segregated from everything else. This author wired the circuit directly to the 8-pin plug of the modem cord. It would also be possible to equip the circuit with male and female 8-pin connectors to connect the computer and the modem cord. Testing the circuit is easy using the following simple BASIC program:

```
1 IF (INP(208) and 32)=0 THEN BEEP
2 GOTO 1
```

When the phone rings, the Model 100 beeps. It is an easy matter to write software in BASIC or machine language to take advantage of the RP signal. First, the CPU selects the modem mode through bit 3 of output port BA. Then it checks that the switches are set to the DIR and ANS positions through bits 4 and 5 of input port BB. After setting the baud rate and word length through a BASIC OPEN MDM: command or through the machine-language routines given in chapter 8, the CPU monitors input port D8 and answers the incoming call. This is done by way of bit 7 of output port BA when bit 5 changes to zero.

The program then interacts with the caller. When the caller hangs up, the incoming carrier would be lost. This would be signaled to the CPU at bit 0 of input port D8, and the CPU disconnects the call.

**Figure 17.3.** Schematic of a ring/detect circuit

Design	Catalog No.	Price	Description
C1,C2	272-1053	.59	0.1 uf 250V capacitor
D1,D2	276-1103	2/.69	1A 400V diode
IC1	276-1654	5/1.98	optoisolator (any of the five types will do)
R1	271-027	.19	2.2K resistor
R2	271-034	.19	10K resistor

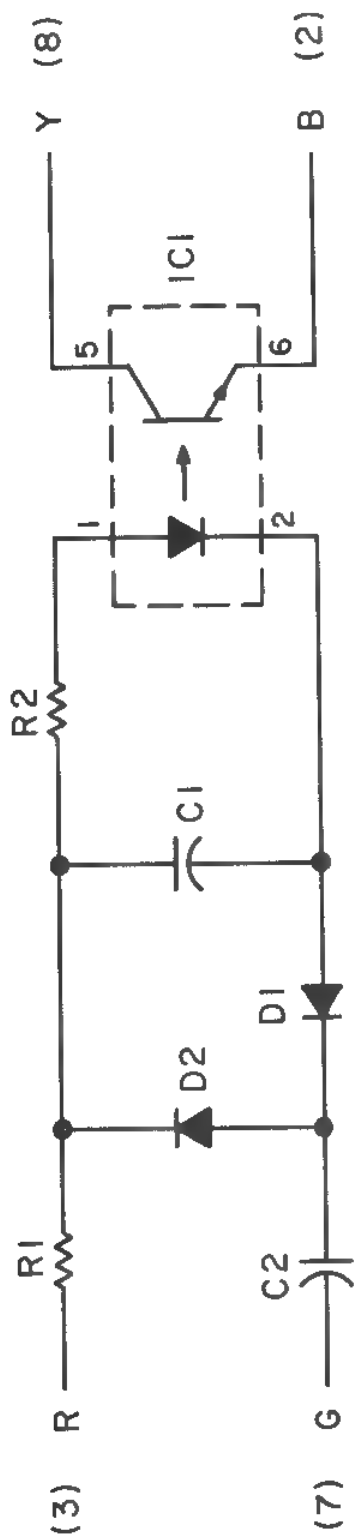


Figure 17.4. Ring/detect circuit



# 18

---

## The Remainder of ROM

---

Although a number of published ROM subroutines have been discussed in earlier chapters, several routines did not fit into those categories. These are discussed here. In addition, other aspects of the ROM operating system are described in this chapter.

### **Published ROM Initialization Routines**

MENU, called at 5797, is not really a subroutine, but rather an entry point for a fundamental ROM process from which there is no return. It goes to the main menu. Jumping to it is preferable to calling it.

The routine IOINIT, start address 6CE0, initializes values at the I/O ports and loads a disk bootstrap loader if a disk is attached. After

the loader is placed at E000 to E0FF, the routine jumps to E000. Assuming no disk is installed, the routine returns to the calling program.

The routine INITIO, called at 6CD6, zeros memory from FF40 to FFFD. This initializes a number of flags, such as XON/ XOFF status, SOUND, baud rate, and the like. Afterwards the routine performs as IOINIT does. Neither of these initialization routines destroys user files.

### PUBLISHED RAM FILE-HANDLING ROUTINES

User files reside in RAM starting at the lowest installed RAM address, with BA files at the bottom, followed by DO files and CO files. The directory is located in high memory from F962 to FA74. There is room to store sixteen filenames, with eleven bytes of information each. The format of the filename information is given in table 18.1.

**Table 18.1.** RAM Directory Format

Byte	Description
0	Directory flag*
1-2	File start address
3-10	Eight-byte file name

**Table 18.2.** RAM directory flag information

Bit	Description
7	0 if a killed file
6	1 if a DO file
5	1 if a CO file
4	1 if located in ROM
3	1 for invisible file
2-1	Reserved
0	Internal use only

Notice that no entry exists for filesize. To determine filesize, either of two techniques may be used. By comparing the various start addresses, one can determine the next file's starting address. The two start addresses may simply be subtracted.

---

\* The directory flag contains the information described in table 18.2

However, the filenames in the directory are not in order by start address, so one must search the entire directory to find the next file up. The ending address of the top file is stored at FBB2+.

Another way to determine file size is by examining the files themselves, since the file itself always contains enough information to determine its size. With BA and DO files, you must scan the file, beginning at the start address, to locate the end-of-file marker. For DO files, the EOF character is 26H. For BA files, it is a three character sequence, 00H.

For CO files, there is no end-of-file character as such, but the length of the file is determined by adding the value of the third byte to the product of 256 times the fourth byte plus six.

## Suzuki and Hayash

Sooner or later you will stumble across the names Suzuki and Hayashi in the directory. Hayashi is the PASTE buffer. Suzuki is the location of the BASIC program, if any, that has not yet been SAVE'd and therefore does not have a name. (It is the file you sometimes see listed as BASIC\*.) Sometimes you select BASIC from the main menu and find that some lines are still present from the last time you were in BASIC; these lines were hiding in Suzuki.

Suzuki and Hayashi take up memory, of course, like any other file. It can be emptied by entering BASIC and typing NEW.

## DO Files

A DO file's first character is that which would be returned if you accessed it with TEXT or with BASIC's OPEN and INPUT statements. The file ends with the 26H described above.

## BA File Format

A BA file in RAM is set up as a sequence of lines, each with the following form:

- 2 bytes            Hex address of line number to follow
- 2 bytes            Current line number (hex)
- numerous bytes   Program line in tokenized form
- 1 byte             Null (00H)

The file ends with two more nulls (00H), making the last three characters nulls.

## CO File Format

The format of a CO file is as follows:

- 2 bytes Address to load file to 5
- 2 bytes Number of bytes to load (six fewer than the size of the RAM file, because of these six address bytes) 4094
- 2 bytes Transfer (start) address
- many bytes Contents of file

## ACCESSING FILES

BASIC's PEEK statement can be used to examine file contents without the danger of altering the file. The use of POKEs with files, however, is to be discouraged unless you know precisely what you are doing. If you must POKE, keep the following in mind.

- POKE only within the files, not elsewhere in RAM. In particular, do not POKE to addresses at or above 62960 (F5F0).
- If you create an end-of-file (EOF) marker within a DO file, you must delete any characters from there up to the beginning of the next file. Use MASDEL, described below.
- If, within a BA file, you tamper with any of the file format addresses described above in memory, such as the BASIC address of the following line number, the program will no longer function properly.

## Published ROM Calls for Manipulating RAM Files

Most of the following routines can be used only to manipulate DO files in RAM.

The routine MAKTXT, called at 220F, creates a DO file in the RAM directory with the name specified in the buffer at FC93 through FC98. If the file already exists, the routine returns with the carry flag set.

Upon exiting the routine, the HL register points to the TOP address of the new file (the address at which new characters would be added), and DE points to the address of the directory file flag, located somewhere in the directory starting at F962.

The routine CHKDC, called at 5AA9, examines the directory to determine if a specified DO filename is present in the directory as a valid file. Prior to the call, DE should point to the first in a series of memory addresses (RAM or ROM) containing a filename in ASCII followed by a null.

Upon return the Z flag is set if no such file was found. If the file was found, HL points to the start, or lowest, address of the file.

The routine GTXTTB, called at 5AE3, finds the TOP address of the file assuming its location in the directory is known. If you provide the address of the directory entry in HL, it returns the TOP address of the file.

The routine KILASC, called at 1FBE, kills a DO file. Prior to the call, DE must be set to the starting address of the file, and HL must be set to the address of the directory entry. Everything above it in user memory (CO files and other DO files) is moved down to fill in the space.

The routine INSCHR, called at 6B61, inserts one character in a DO file. Prior to the call, HL points to the address at which to insert the character, and A contains the character to be inserted. Everything above in user area (all CO files and some DO files) is moved up one position.

If there is no more room in RAM, the routine returns with the carry flag set.

The routine MAKHOL, called at 6B6D, inserts a specified number of spaces in a file. Everything is moved up in memory to accommodate the spaces. These spaces may later be changed as desired. Prior to the call, BC must contain the number of spaces to be inserted, and HL must point to the address at which to insert the spaces. HL and BC are preserved, which is handy. If there was insufficient room in RAM, the routine returns with the carry flag set.

The routine MASDEL, called at 6B9F, deletes a specified number of characters from a file. Prior to the call, BC must contain the number



of characters to be deleted, and HL should point to the address at which to begin deleting. HL and BC are preserved.

## Block Moves

The Model 100 ROM contains a number of block move routines. In each case, a source address is loaded to a register pair, a destination address is loaded to another register pair, and the number of bytes to be transferred is loaded to another register (for a transfer of up to 256 bytes) or to another register pair (for a transfer of up to 65536 bytes). At this point, the routine can either load and increment or load and decrement, as shown in table 18.3.

**Table 18.3.** Block move subroutines

Source	Destination	# of Bytes Bytes	Increment or Decrement	Call Address
BC	DE	L	Increment	290F
DE	HL	A	Increment	5A62
DE	HL	B	Increment	3469
DE	HL	B	Decrement	3472
HL	DE	B	Increment	2542
HL	DE	BC	Decrement	6BE6
HL	DE	BC	Increment	6BDB
HL	DE	C	Decrement	2EE6
HL	DE	till 0 is loaded	Increment	65C3

One routine, at 65C3, is different in that it loads until such time as a null character is encountered in the source location.

## Lowercase Conversion

The routine at 0FE8 converts lowercase to uppercase. Prior to the call, HL must point to a character. After the call, the accumulator contains that character, converted to uppercase.

### CONVERTING NUMERICAL HEX TO ASCII

The routine at 1999 converts the numerical hex byte pointed to by DE to ASCII, with the result in HL. Then DE and HL are incremented. (All the routine does is OR the contents with 30H.) No error checking is undertaken. The value at DE must be in the range of 0 to 9.

## CONVERTING TWO NUMERICAL HEX BYTES TO ASCII

The routine at 1996 causes the previous routine to be executed twice. This is useful for converting seconds from the clock/calendar.

## REGISTER-PAIR COMPARISON

The subroutine called by RST 3 (RST 18) compares DE to HL. If they are equal, the zero flag will be set.

## UTILITY FOR COMMAND DECODING

The subroutine at 6CA9 is a useful general-purpose command decoder. Prior to the call, DE points to a command table containing four-letter commands and jump addresses. HL points to a command received as user input. The command is converted to upper case and is compared one-by-one with the commands in the table. If there is a match, the routine jumps to the address in the table associated with that command. If there is no match, the routine returns with the Z flag set.

The table is composed of one or more command entries, followed by a null. Each command entry is composed of four letters and a two-byte jump address.

## RAM VARIABLE MAP

A variety of variables are located in high-end RAM. They are listed in table 18.4. Many of these may be changed by a user program as needed.

**Table 18.4.** RAM variables

Address	Description	Initialized Where
F5F0	Start of system	035A
F5F2	SP upon power-down	035C
F5F4+	HIMEM value	035E
F5F6	Power-on hook	0360
F5F9	BCR interrupt hook	0363
F5FC	UART DR hook	0366
F5FF	TP interrupt hook	0369
F602	LPS interrupt hook	036C

Continued on following page

F605	Power-on hook	036F
F60F	Option ROM hook	0379
F62A	Option ROM installed	0394
F62B	20 or 10 pps	
F639	Cursor row	03A3
F63A	LCD cursor column	03A4
F63B	LCD active lines	03A5
F63C	LCD active columns	03A6
F63D	Label line flags	03A7
F648	Reverse video if 1	03B2
F64E	X-pixel set	03B8
F64F	Y-pixel set	03B9
F657	Power-down value	03C1
F65B-F660	Initial Stat setting	03C5
F674	LPOS printer column	03DE
F675	Output 0=LCD 1=LP	03DF
F678	Top of available RAM	03E2
F685	Keyboard buffer	
F788	POS- cursor position	
F789	Key labels	
F88C+	Location of PASTE buffer	
F923	Seconds— units	
F924	Seconds— tens	
F925	Minutes— units	
F926	Minutes— tens	
F927	Hours— units	
F928	Hours— tens	
F929	Day of month-units	
F92A	Day of month-tens	
F92B	Day of week	
F92C	Month	
F92D	Year-units	
F92E	Year-tens	
F962-F9B9	Directory 6BF1+	
F9BA-F9C4	Option ROM or filename	
F9C5-FA74	Directory	
FAAC	Character recently sent	
FAAE	Port A8 contents	
FAC0	Lowest installed RAM	
FADA-FB14	RST 38 hooks-all RET	
FB16-FB39	RST 38 hooks-all error 5	
FBB0+	Top of DO files	
FBB2+	Top of CO files	
FC18	FLoating Point Accumulator 1	
FC69	Floating Point Accumulator 2	
FC82	Maxfiles?	
FC93-FC98	Filename	

Continued on following page

FE00-FF3F	LCD memory	
FF40	Distant computer sent XOFF	
FF41	This Model 100 sent XOFF	
FF42	Is XOFF enabled?	
FF44	SOUND OFF=1 ON=0	
FF45	Port E8 contents	
FF8B	Address of baud rate	



## Appendix A. Nonprintable Characters and Model 100 Assignments

The following table indicates the meanings that other devices may give to nonprintable ASCII characters. It also shows what meaning, if any, the Model 100 assigns to these characters.

Decimal	ASCII Meaning	Model 100 Meaning
0	Null	
1	SOH-Start of Heading	
2	STX-Start of Text	
3	ETX-End of Text	
4	EOT-End of Transmission	
5	ENQ-Enquiry	
6	ACK-Acknowledge	
7	BEL-Bell	produces BEEP
8	BS-Backspace	moves cursor one space to left
9	HT-horizontal tab	moves cursor to next 8th column
10	LF-line feed	moves cursor one line down (no horizontal movement)
11	VT-vertical tab	cursor to home
12	FF-form feed	clears screen
13	CR-carriage return	cursor to left edge
14	SO-shift out	
15	SI-shift in	
16	DLE-data link escape	
17	DC1-device control 1	if XON/XOFF enabled, continue transmission
18	DC2-device control 2	
19	DC3-device control 3	if XON/XOFF enabled, pause in transmission
20	DC4-device control 4	
21	NAK-negative acknowledge	
22	SYN-synchronous idle	
23	ETB-end of transmission block	
24	CAN-cancel	
25	EM-end of medium	
26	SUB-substitute	
27	ESC-escape	(see table 13.4)

## 282 Inside the TRS-80 Model 100

---

28	FS-file separator	
29	GS-group separator	
30	RS-record separator	
31	US-unit separator	
127	DEL-delete	same as backspace

## Appendix B. ROM Map

This map will be of help to those who are disassembling ROM routines.

Address	Function
0000	hardware reset address
0008	restart 08
0010	restart 10
0018	restart 18-compares HL and DE
0020	restart 20-character to LCD or LPT
0024	low-power signal routine
0028	restart 28-check variable type
002C	bar code reader routine
0030	restart 30-sign of floating point number
0034	UART data ready routine
0038	restart 38-ram hooks
003C	clock/calendar pulse routine
0040 to 007F	address table for BASIC functions SGN to MID\$, keywords of which appear at 01F0 to 025F
0080 to 018E	BASIC command keywords END to NEW, with address table starting at 0262
018F to 01D5	BASIC function keywords TAB to STEP
01D6 to 01EF	BASIC operator keywords "+" to "<", with address table starting at 02F8
01F0 to 025F	BASIC function keywords SGN to MID\$, with address table starting at 0040
0262 to 02E1	address table for BASIC command keywords END to NEW, located at 0080 to 018E
02EE to 031B	address table for BASIC operators, used at 10DA
031C to 0359	BASIC two-character error messages
035A to 03E9	initialization RAM image loaded to F5F0-F76F
035C	initialization value for SP upon power-down
035E	initialization value for HIMEM
0360	initialization value for power-on hook
0363	initialization value for bar code reader interrupt hook
0366	initialization value for UART data ready interrupt hook
0369	initialization value for clock/calendar interrupt hook



Address	Function
036C	initialization value for low power signal interrupt hook
036F	power-on routine— option ROM testing
0394	initialization value for option ROM presence flag
03A3 to 03A6	initialization values for screen position and size parameters
03A7	initialization value for label line flag
03B2	initialization value for reverse video flag
03C1	initialization value for power-down countdown
03C5	initialization value for Stat
03DE	initialization value for LPOS
03DF	initialization value for output flag
04DD	prints error message based on A
0726	BASIC command FOR
076B	BASIC command TO
0783	BASIC command STEP
0840	BASIC command dispatcher based on token in A
0858	sets pointer to BASIC text
0872	BASIC command DEF
0881	BASIC command DEFDBL
0886	BASIC command DEFINT
0896	BASIC command DEFSNG
089F	BASIC command DEFSTR
08DB	FC error
08EB	convert ASCII decimal to hex in DE
090F	BASIC command RUN
091E	BASIC command GOSUB
0936	BASIC command GOTO
0966	BASIC command RETURN
099E	BASIC command DATA
09A0	BASIC command ELSE, REM
09C3	BASIC command LET
0A2F	BASIC command ON
0A34	BASIC command ON ERROR
0AB0	BASIC command RESUME
0B0F	BASIC command ERROR
0B1A	BASIC command IF
0B4E	BASIC command LPRINT
0B56	BASIC command PRINT
0C01	BASIC command TAB(
0C45	BASIC command LINE
0C50	BASIC command LINE INPUT
0C74	"Redo from start"
0C99	BASIC command INPUT#

Address	Function
0CA3	BASIC command INPUT
0CD9	BASIC command READ
0D71	BASIC command "Extra ignored"
0F47	BASIC function ERR
0F56	BASIC function ERL
0F7E	BASIC function VARPTR
0FE8	converts LC to UC
1054	BASIC operator NOT
108C	BASIC operator OR
1097	BASIC operator AND
10A2	BASIC operator XOR
10AD	BASIC operator EQV
10B5	BASIC operator IMP
10C8	BASIC function LPOS
10CE	BASIC function POS
1100	BASIC function INP
110C	BASIC command OUT
112E	convert ASCII to integer
1138	BASIC command LLIST
1140	BASIC command LIST
11A2	print from buffer till zero byte reached
11AA	put data in buffer until zero byte reached
1284	BASIC function PEEK
128B	BASIC command POKE
12CB	wait for character from keyboard
12F0	paste routine
13A5	toggle label line
13DB	check keybaord queue for characters
1412	break routine
1419	POWER routine
1431	power down sequence
1451	POWER OFF
1459	BASIC command POWER CONT
1469	set power down value
1470	print character without expanding ASCII 09
148A	read cassette header and synch byte
14A8	motor on
14AA	motor off
14B0	read a character from cassette
14C1	send character to cassette and update checksum
14D2	LCD device control block
14D8	LCD file open
14E5	LCD file put
14F2	CRT device control block
14FC	RAM device control block

Address	Function
1506	RAM file open
158D	RAM file close
167F	CAS device control block
1689	CAS file open
16AD	CAS file close
16C7	CAS file put
16D2	CAS file get
1754	LPT device control block
175A	LPT file put
1762	COM device control block
176C	MDM file open
176D	COM file open
179E	COM file close
17A8	COM and MDM file put
17B0	COM and MDM file get
17D1	MDM device control block
17DB	MDM file close
17E6	sets serial interface per Stat
1877	WAND device control block
1889	BASIC function EOF
1904	BASIC function TIME\$
190F	get time string to HL
1924	BASIC function DATE\$
192F	get date string to HL
1955	BASIC function DAY\$
1962	get day string to HL
1978	"SunMonTueWedThuFriSat"
1999	convert hex to digit
19A0	loads clock/calendar into RAM
19AB	BASIC command TIME\$=
19BD	BASIC command DATE\$=
19F1	BASIC command DAY\$=
19FA	BASIC command MAXRAM
1A78	BASIC command IPL
1A9E	BASIC command MDM, COM
1AB2	BASIC command KEY(
1B0F	BASIC command ON TIME\$
1B32	clock/calendar pulse handler
1BB8	BASIC keyword KEY
1BBD	BASIC command KEY LIST
1BE0	prints memory to screen filtering
1C57	BASIC command PSET
1C66	BASIC command PRESET
1D90	BASIC function CSRLIN
1D9B	BASIC keyword MAX
1DB2	BASIC command MAXFILES

Address	Function
1DB9	BASIC keyword HIMEM
1DC3	BASIC command WIDTH
1DC5	BASIC command SOUND
1DE5	BASIC command SOUND OFF
1DE6	BASIC command SOUND ON
1DEC	BASIC command MOTOR
1DFA	BASIC command CALL
1E22	BASIC command SCREEN
1E5E	BASIC command LCOPY
1E5E	screen dump routine
1F3A	BASIC command FILES
1F91	BASIC command KILL
1FBE	BASIC command KILL for DO file
2037	BASIC command NAME
20FE	BASIC command NEW
220F	opens RAM DO file
2280	BASIC command CSAVE
22B9	block move to tape
22CC	CSAVEM
22DD	BASIC command CSAVEM
2377	BASIC command CLOAD
2413	block move from tape
2456	CLOAD?
2491	BASIC command LOADM
2491	BASIC command RUNM
24A7	BASIC command CLOADM
2542	block move of B bytes from HL to DE increasing
2573	CLOADM?
25D5	"Top: "
25DB	"End: "
25E1	"Exe: "
260B	open for output CAS:.BA
260E	open for output CAS:.DO
2611	open for output CAS:.CO
2650	open for input CAS:.BA
2653	open for input CAS:.DO
2656	open for input CAS:.CO
26FE	"Found:"
2705	"Skip:"
273A	BASIC function STR\$
27B1	prints a line to screen
28CC	string concatenation
290C	block move bytes from BC to DE increasing
2943	BASIC function LEN
294F	BASIC function ASC
295F	BASIC function CHR\$
296D	BASIC function STRING\$

Address	Function
298E	BASIC function SPACE\$
29AB	BASIC function LEFT\$
29D6	BASIC function RIGHT\$
29E6	BASIC function MID\$
2A07	BASIC function VAL
2A37	BASIC function INSTR
2B4C	BASIC function FRE
2B69	BASIC operation double precision subtraction
2B78	BASIC operation double precision addition
2CFF	BASIC operation double precision multiplication
2DC7	BASIC operation double precision division
2EE6	block move
2EEF	BASIC function COS
2F09	BASIC function SIN
2F58	BASIC function TAN
2F71	BASIC function ATN
2FCF	BASIC function LOG
305A	BASIC function SQR
30A4	BASIC function EXP
313E	BASIC function RND
325C to 33DB	floating point constants
33DC	restart 30 routine
33F2	BASIC function ABS
3407	SGN
3469	block move B bytes from DE to HL increasing
3472	block move B bytes from DE to HL decreasing
3498	BASIC operator single precision comparison
34C2	BASIC operator integer comparison
34FA	BASIC operator double precision comparison
3501	BASIC function CINT
352A	BASIC function CSNG
35BA	BASIC function CDBL
3645	BASIC function FIX
3654	BASIC function INT
36F8	BASIC operator integer subtraction
3704	BASIC operator integer add tion
3725	BASIC operator integer multiplication
377E	BASIC operator integer division
37DF	BASIC operator MOD
37F4	BASIC operator single precision addition
37FD	BASIC operator single precision subtraction
3803	BASIC operator single precision multiplication
380E	BASIC operator single precision division
39D4	convert hex to integer and print
3D7F	BASIC operator single precision exponentiation

Address	Function
3D8E	BASIC operator double precision exponentiation
3DF7	BASIC operator integer exponentiation
3FA0	BASIC command TIME\$ ON
3FB2	BASIC command TIME\$ OFF
3FB9	BASIC command TIME\$ STOP
407F	BASIC command RESTORE
409A	BASIC command STOP
409F	BASIC command END
40DA	BASIC command CONT
40F9	BASIC command CLEAR
4174	BASIC command NEXT
4222	CR and LF to LCD
4225	LF to LCD
4229	BEEP to LCD
422D	LCD cursor home
4231	LCD clear screen
4235	LCD set label line
423A	LCD unlock label line
423F	LCD disallow scrolling
4244	LCD allow scrolling
4249	LCD cursor on
424E	LCD cursor off
4253	LCD delete line at cursor
4258	LCD insert blank line at cursor
425D	LCD erase to end of line
4262	LCD escape X sequence
4269	LCD enter reverse character mode
426E	LCD exit reverse character mode
4270	LCD send escape sequence
427C	LCD set cursor position
428A	LCD erase function key display
42A5	LCD set and display function keys
42A8	LCD display function keys
438A to 43A1	lookup table for special ASCII LCD characters
43B8 to 43F9	lookup table for LCD escape sequences
4408	LCD TAB routine
4431	LCD ESC p routine
4433	LCD ESC q routine
4437	LCD ESC U routine
4438	LCD ESC T routine
443F	LCD ESC V routine
4441	LCD ESC W routine
444A	LCD ESC X routine
4453	LCD ESC C routine
445C	LCD ESC D routine
4461	LCD backspace routine

Address	Function
4469	LCD ESC A routine
446E	LCD ESC B routine
4480	LCD TAB routine
4494	LCD line feed routine
44A8	LCD vertical tab routine
44A8	LCD ESC H routine
44AF	LCD ESC P routine
44BA	LCD ESC Q routine
44C4	LCD ESC M routine
44EA	LCD ESC L routine
4535	LCD ESC I routine
4537	LCD ESC k routine
4548	LCD ESC J routine
4548	LCD ESC E routine
463E	prompt with ? and get line
4644	get line from keyboard
4684	BASIC control-C handler
4696	BASIC ENTER handler
46A0	BASIC backspace handler
46A0	BASIC leftarrow handler
46A0	BASIC CTRL-H handler
46C3	BASIC CTRL-U handler
46C3	BASIC CTRL-X handler
46CA	BASIC TAB handler
478B	BASIC command DIM
4991	BASIC command USING
4B44	character to LCD
4B55	put to printer expanding ASCII 09
4BA0	carriage return to printer
4BEA	BASIC function INKEY\$
4C0F	filename string scan
4CCB	BASIC command OPEN
4D59	COM file close routine
4D70	BASIC command LOAD
4D71	BASIC command MERGE
4DCF	BASIC command SAVE
4E28	BASIC command CLOSE
4E8E	BASIC function INPUT\$
4F0A	clear B bytes of memory at HL
4F0B	load A into B bytes of memory at HL
404E	bad file name
5051	already open
5054	direct statement in file
5057	file not found

Address	Function
505A	file not open
505D	bad file number
5060	undefined input error IE
5063	input past end
5066	undefined error FL
506B	BASIC function LOF
506D	BASIC function LOC
506F	BASIC command LFILES
5071	BASIC function DSKO\$
5073	BASIC function DSKI\$
50F1 to 5112	BASIC device control block name match table
5113 to 5124	address table for device control blocks
5146	TELCOM start location
51C0	TELCOM Stat function key
522F	TELCOM Call function key
524D	TELCOM Find function key
52BB	disconnect phone line
52D0	connect phone line
532D	autodialer routine
5455	TELCOM Term function key
5523	TELCOM Prev function key
553E	TELCOM Full/Half function key
5550	TELCOM Echo function key
559D	TELCOM Up function key
567E	TELCOM Down function key
571E	TELCOM Bye function key
5791	CR/LF, then display string at HL up to null
5797	go to main menu
57DF	display copyright notice
582E	display number of free bytes
5834	menu select handler
5970	list files on screen
59C9	move cursor across filenames
5A58	send string to screen
5A62	block move
5A79	clear function keys
5A7C	set function keys
5A9E	display function keys
5AA9	search for filename in directory
5AE3	get top address of file
5B3E	used to clear all function keys
5B46	normal BASIC function keys
5B68	ADDRSS routine
5B6F	SCHEDL routine



Address	Function
5BF5	ADDRSS/SCHEDL Find function key
5BF7	ADDRSS/SCHEDL Lfind function key
5D46	is character at HL a space?
5D6A	home cursor
5DEE	TEXT routine
5E51	BASIC command EDIT
6016 to 6055	TEXT CTRL-character address table
607C	TEXT CTRL-P (escape) handler
608A	TEXT TAB handler
60BE	TEXT carriage return handler
60DE	TEXT rightarrow handler
60E2	TEXT downarrow handler
610B	TEXT backspace handler
6118	TEXT delete handler
6151	TEXT leftarrow handler
6155	TEXT uparrow handler
617A	TEXT SHIFT-rightarrow handler
618C	TEXT SHIFT-leftarrow handler
61C2	TEXT SHIFT-uparrow handler
61D7	TEXT SHIFT-downarrow handler
61FD	TEXT CTRL-rightarrow handler
620B	TEXT CTRL-leftarrow handler
6210	TEXT CTRL-uparrow handler
621C	TEXT CTRL-downarrow handler
6242	TEXT Sel function key handler
628F	TEXT CTRL-C and SHIFT-BREAK handler
6431	TEXT Copy function key handler
6445	TEXT Cut function key handler
6551	TEXT Find function key handler
65C3	block move HL to DE until null
667C	TEXT tab handler
6691	TEXT SHIFT-PRINT handler
6713	TEXT Save function key handler
6774	TEXT Load function key handler
6AC3	HALT in TEXT- why?
6B61	insert character in RAM file
6B6D	insert spaces in RAM file
6B9F	delete characters from RAM file
6BDB	block move BC bytes from HL to DE increasing
6BE6	block move BC bytes from HL to DE decreasing
6BF1 to 6C48	initialization value for RAM directory
6C49	BASIC routine starts here

Address	Function
6CD6	cold start reset
6CE0	warm start reset
6CED	set 256 Hertz interrupt
6CFC	initialize disk drive
6D3F	put to LPT
6D6D	inspect incoming serial queue
6D7E	get incoming serial data
6DAC	handles UART data ready
6DBE	parity, overrun, framing error
6E0B	put CTRL-Q to UART
6E1E	put CTRL-S to UART
6E32	put byte to UART
6E75	set UART baud rate
6E94 to 6EA5	baud rate table
6EA6	set baud rate and modem
6ECB	deactivate UART
6EEF	carrier detect
6F31	enable CTL-S, CTL-Q protocol
6F32	disable CTL-S, CTL-Q protocol
6F46	cassette write header and synch byte
6F5B	put one byte to cassette
6F85	cassette read header and synch byte
6FDB	cassette bit-input routine
702A	cassette byte-input routine
7055 to 7241	keyboard scanning code
71E4	add a character to keyboard buffer
7242	get keyboard contents
7270	check keyboard for characters or SHIFT-BREAK
7283	check keyboard for SHIFT-BREAK or CTRL-C
729F	check keyboard for SHIFT-BREAK
72C5	BASIC command SOUND
7304 to 7326	disk input routine
7329	clock/calendar data loader
7391	clock/calendar timing pulse handler
7440	cursor set routine
744C	pixel set routine
744D	pixel reset routine
7551 to 7653	LCD location table
7662	BEEP routine
7676	toggle beeper
767D to 76DB	disk bootstrap loader
76DE	disk output routine
7711	LCD character generation table

<b>Address</b>	<b>Function</b>
7BF1 to 7D32	keyboard decoding table
7D33	RESET routine
7D43	erase all files if CTRL-BREAK pushed
7D6C	test for option ROM
7DE7	erase all files
7E24	load option ROM if present
7EAC	prints number of free bytes
7EC6	initialize RAM vectors
7FD6	restart 38 handler

### Appendix C. 8080, 8085, & Z80 Opcodes

Decimal	Hex	8080 Opcode	Z80 Opcode
0	00	NOP	NOP
1	01 FF FF	LXI B,FFFF	LD BC,FFFF
2	02	STAX B	LD (BC),A
3	03	INX B	INC BC
4	04	INR B	INC B
5	05	DCR B	DEC B
6	06 FF	MVI B,FF	LD B,FF
7	07	RLC	RLCA
8	08	-Data-	-Data-
9	09	DAD B	ADD HL,BC
10	0A	LDAX B	LD A,(BC)
11	0B	DCX B	DEC BC
12	0C	INR C	INC C
13	0D	DCR C	DEC C
14	0E FF	MVI C,FF	LD C,FF
15	0F	RRC	RRCA
16	10	-Data-	-Data-
17	11 FF FF	LXI D,FFFF	LD DE,FFFF
18	12	STAX D	LD (DE),A
19	13	INX D	INC DE
20	14	INR D	INC D
21	15	DCR D	DEC D
22	16 FF	MVI D,FF	LD D,FF
23	17	RAL	RLA
24	18	-Data-	-Data-
25	19	DAD D	ADD HL,DE
26	1A	LDAX D	LD A,(DE)
27	1B	DCX D	DEC DE
28	1C	INR E	INC E
29	1D	DCR E	DEC E
30	1E FF	MVI E,FF	LD E,FF
31	1F	RAR	RRA

Table C.1. Numerical list of Opcodes

Decimal	Hex	8080 Opcode	Z80 Opcode
32	20	RIH	-----
33	21 FF FF	LXI H,FFFF	LD HL,FFFF
34	22 FF FF	SHLD FFFF	LD (FFFF),HL
35	23	INX H	INC HL
36	24	INR H	INC H
37	25	DCR H	DEC H
38	26 FF	MVI H,FF	LD H,FF
39	27	DAA	DAA
40	28	-Data-	-Data-
41	29	DAD HL	ADD HL,HL
42	2A FF FF	LHLD FFFF	LD HL,(FFFF)
43	2B	DCX H	DEC HL
44	2C	INR L	INC L
45	2D	DCR L	DEC L
46	2E FF	MVI L,FF	LD L,FF
47	2F	CMA	CPL
48	30	SIM	-----
49	31 FF FF	LXI SP,FFFF	LD SP,FFFF
50	32 FF FF	STA FFFF	LD (FFFF),A
51	33	INX SP	INC SP
52	34	INR M	INC (HL)
53	35	DCR M	DEC (HL)
54	36 FF	MVI M,FF	LD (HL),FF
55	37	STC	SCF
56	38	-Data-	-Data-
57	39	DAD SP	ADD HL,SP
58	3A FF FF	LDA FFFF	LD A,(FFFF)
59	3B	DCX SP	DEC SP
60	3C	INR A	INC A
61	3D	DCR A	DEC A
62	3E FF	MVI A,FF	LD A,FF
63	3F	CMC	CCF
64	40	MOV B,B	LD B,B
65	41	MOV B,C	LD B,C

Table C.1. (cont.)

Decimal	Hex	8080 Opcode	Z80 Opcode
66	42	MOV B,D	LD B,D
67	43	MOV B,E	LD B,E
68	44	MOV B,H	LD B,H
69	45	MOV B,L	LD B,L
70	46	MOV B,M	LD B,(HL)
71	47	MOV B,A	LD B,A
72	48	MOV C,B	LD C,B
73	49	MOV C,C	LD C,C
74	4A	MOV C,D	LD C,D
75	4B	MOV C,E	LD C,E
76	4C	MOV C,H	LD C,H
77	4D	MOV C,L	LD C,L
78	4E	MOV C,M	LD C,(HL)
79	4F	MOV C,A	LD C,A
80	50	MOV D,B	LD D,B
81	51	MOV D,C	LD D,C
82	52	MOV D,D	LD D,D
83	53	MOV D,E	LD D,E
84	54	MOV D,H	LD D,H
85	55	MOV D,L	LD D,L
86	56	MOV D,M	LD D,(HL)
87	57	MOV D,A	LD D,A
88	58	MOV E,B	LD E,B
89	59	MOV E,C	LD E,C
90	5A	MOV E,D	LD E,D
91	5B	MOV E,E	LD E,E
92	5C	MOV E,H	LD E,H
93	5D	MOV E,L	LD E,L
94	5E	MOV E,M	LD E,(HL)
95	5F	MOV E,A	LD E,A
96	60	MOV H,B	LD H,B
97	61	MOV H,C	LD H,C

Table C.1. (cont.)

Decimal	Hex	8080 Opcode	Z80 Opcode
98	62	MOV H,D	LD H,D
99	63	MOV H,E	LD H,E
100	64	MOV H,H	LD H,H
101	65	MOV H,L	LD H,L
102	66	MOV H,M	LD H,(HL)
103	67	MOV H,A	LD H,A
104	68	MOV L,B	LD L,B
105	69	MOV L,C	LD L,C
106	6A	MOV L,D	LD L,D
107	6B	MOV L,E	LD L,E
108	6C	MOV L,H	LD L,H
109	6D	MOV L,L	LD L,L
110	6E	MOV L,M	LD L,(HL)
111	6F	MOV L,A	LD L,A
112	70	MOV M,B	LD (HL),B
113	71	MOV M,C	LD (HL),C
114	72	MOV M,D	LD (HL),D
115	73	MOV M,E	LD (HL),E
116	74	MOV M,H	LD (HL),H
117	75	MOV M,L	LD (HL),L
118	76	HLT	HALT
119	77	MOV M,A	LD (HL),A
120	78	MOV A,B	LD A,B
121	79	MOV A,C	LD A,C
122	7A	MOV A,D	LD A,D
123	7B	MOV A,E	LD A,E
124	7C	MOV A,H	LD A,H
125	7D	MOV A,L	LD A,L
126	7E	MOV A,M	LD A,(HL)
127	7F	MOV A,A	LD A,A
128	80	ADD B	ADD A,B

Table C.1. (cont.)

Decimal	Hex	8080 Opcode	Z80 Opcode
129	81	ADD C	ADD A,C
130	82	ADD D	ADD A,D
131	83	ADD E	ADD A,E
132	84	ADD H	ADD A,H
133	85	ADD L	ADD A,L
134	86	ADD M	ADD A,(HL)
135	87	ADD A	ADD A,A
136	88	ADC B	ADC A,B
137	89	ADC C	ADC A,C
138	8A	ADC D	ADC A,D
139	8B	ADC E	ADC A,E
140	8C	ADC H	ADC A,H
141	8D	ADC L	ADC A,L
142	8E	ADC M	ADC A,(HL)
143	8F	ADC A	ADC A,A
144	90	SUB B	SUB A,B
145	91	SUB C	SUB A,C
146	92	SUB D	SUB A,D
147	93	SUB E	SUB A,E
148	94	SUB H	SUB A,H
149	95	SUB L	SUB A,L
150	96	SUB M	SUB (HL)
151	97	SUB A	SUB A,A
152	98	SBB B	SBC A,B
153	99	SBB C	SBC A,C
154	9A	SBB D	SBC A,D
155	9B	SBB E	SBC A,E
156	9C	SBB H	SBC A,H
157	9D	SBB L	SBC A,L
158	9E	SBB M	SBC A,(HL)
159	9F	SBB A	SBC A,A

Table C.1. (cont.)



Decimal	Hex	8080 Opcode	Z80 Opcode
160	A0	ANA B	AND B
161	A1	ANA C	AND C
162	A2	ANA D	AND D
163	A3	ANA E	AND E
164	A4	ANA H	AND H
165	A5	ANA L	AND L
166	A6	ANA M	AND (HL)
167	A7	ANA A	AND A
168	A8	XRA B	XOR B
169	A9	XRA C	XOR C
170	AA	XRA D	XOR D
171	AB	XRA E	XOR E
172	AC	XRA H	XOR H
173	AD	XRA L	XOR L
174	AE	XRA M	XOR (HL)
175	AF	XRA A	XOR A
176	B0	ORA B	OR B
177	B1	ORA C	OR C
178	B2	ORA D	OR D
179	B3	ORA E	OR E
180	B4	ORA H	OR H
181	B5	ORA L	OR L
182	B6	ORA M	OR (HL)
183	B7	ORA A	OR A
184	B8	CMP B	CP B
185	B9	CMP C	CP C
186	BA	CMP D	CP D
187	BB	CMP E	CP E
188	BC	CMP H	CP H
189	BD	CMP L	CP L
190	BE	CMP M	CP (HL)

Table C.1. (cont.)

Decimal	Hex	8080 Opcode	Z80 Opcode
191	BF	CMP A	CP A
192	C0	RNZ	RET NZ
193	C1	POP B	POP BC
194	C2 FF FF	JNZ FFFF	JP NZ,FFFF
195	C3 FF FF	JMP FFFF	JP FFFF
196	C4 FF FF	CNZ FFFF	CALL NZ,FFFF
197	C5	PUSH B	PUSH BC
198	C6 FF	ADI FF	ADD A,FF
199	C7	RST 0	RST 00
200	C8	RZ	RET Z
201	C9	RET	RET
202	CA FF FF	JZ FFFF	JP Z,FFFF
203	CB	-Data-	-Data-
204	CC FF FF	CZ FFFF	CALL Z,FFFF
205	CD FF FF	CALL FFFF	CALL FFFF
206	CE FF	ACI FF	ADC A,FF
207	CF	RST 1	RST 08
208	D0	RNC	RET NC
209	D1	POP D	POP DE
210	D2 FF FF	JNC FFFF	JP NC,FFFF
211	D3 FF	OUT FF	OUT (FF),A
212	D4 FF FF	CNC FFFF	CALL NC,FFFF
213	D5	PUSH D	PUSH DE
214	D6 FF	SUI FF	SUB FF
215	D7	RST 2	RST 10
216	D8	RC	RET C
217	D9	-Data-	-Data-
218	DA FF FF	JC FFFF	JP C,FFFF
219	DB FF	IN FF	IN A,(FF)
220	DC FF FF	CC FFFF	CALL C,FFFF
221	DD	-Data-	-Data-

Table C.1. (cont.)

Decimal	Hex	8080 Opcode	Z80 Opcode
222	DE FF	SBI FF	SBC FF
223	DF	RST 3	RST 18
224	E0	RPO	RET PO
225	E1	POP H	POP HL
226	E2 FF FF	JPO FFFF	JP PO,FFFF
227	E3	XTHL	EX (SP),HL
228	E4 FF FF	CPO FFFF	CALL PO,FFFF
229	E5	PUSH H	PUSH HL
230	E6 FF	ANI FF	AND FF
231	E7	RST 4	RST 20
232	E8	RPE	RET PE
233	E9	PCHL	JP (HL)
234	EA FF FF	JPE FFFF	JP PE,FFFF
235	EB	XCHG	EX DE,HL
236	EC FF FF	CPE FFFF	CALL PE,FFFF
237	ED	-Data-	-Data-
238	EE FF	XRI FF	XOR FF
239	EF	RST 5	RST 28
240	F0	RP	RET P
241	F1	POP PSW	POP AF
242	F2 FF FF	JP FFFF	JP P,FFFF
243	F3	DI	DI
244	F4 FF FF	CP FFFF	CALL P,FFFF
245	F5	PUSH PSW	PUSH AF
246	F6 FF	ORI FF	OR FF
247	F7	RST 6	RST 30
248	F8	RM	RET M
249	F9	SPHL	LD SP,HL
250	FA FF FF	JM FFFF	JP M,FFFF
251	FB	EI	EI
252	FC FF FF	CM FFFF	CALL M,FFFF

Table C.1. (cont.)

Decimal	Hex	8080 Opcode	Z80 Opcode
253	FD	-Data-	-Data-
254	FE FF	CPI FF	CP FF
255	FF	RST 7	RST 38

Table C.1. (cont.)

Decimal	Hex	8085 Opcode	8080 Opcode
206	CE FF	ACI FF	ADC A,FF
143	8F	ADC A	ADC A,A
136	88	ADC B	ADC A,B
137	89	ADC C	ADC A,C
138	8A	ADC D	ADC A,D
139	8B	ADC E	ADC A,E
140	8C	ADC H	ADC A,H
141	8D	ADC L	ADC A,L
142	8E	ADC M	ADC A,(HL)
135	87	ADD A	ADD A,A
128	80	ADD B	ADD A,B
129	81	ADD C	ADD A,C
130	82	ADD D	ADD A,D
131	83	ADD E	ADD A,E
132	84	ADD H	ADD A,H
133	85	ADD L	ADD A,L
134	86	ADD M	ADD A,(HL)
198	C6 FF	ADI FF	ADD A,FF
167	A7	ANA A	AND A
160	A0	ANA B	AND B
161	A1	ANA C	AND C
162	A2	ANA D	AND D
163	A3	ANA E	AND E
164	A4	ANA H	AND H
165	A5	ANA L	AND L
166	A6	ANA M	AND (HL)
230	E6 FF	ANI FF	AND FF
205	CD FF FF	CALL FFFF	CALL FFFF
220	DC FF FF	CC FFFF	CALL C,FFFF
252	FC FF FF	CM FFFF	CALL M,FFFF

Table C.2. Alphabetical list of 8085 opcodes

Decimal	Hex	8085 Opcode	Z80 Opcode
47	2F	CMA	CPL
63	3F	CNC	CCF
191	BF	CMP A	CP A
184	B8	CMP B	CP B
185	B9	CMP C	CP C
186	BA	CMP D	CP D
187	BB	CMP E	CP E
188	BC	CMP H	CP H
189	BD	CMP L	CP L
190	BE	CMP M	CP (HL)
212	D4 FF FF	CNC FFFF	CALL NC,FFFF
196	C4 FF FF	CNZ FFFF	CALL NZ,FFFF
244	F4 FF FF	CP FFFF	CALL P,FFFF
236	EC FF FF	CPE FFFF	CALL PE,FFFF
254	FE FF	CPI FF	CP FF
228	E4 FF FF	CPO FFFF	CALL PO,FFFF
204	CC FF FF	CZ FFFF	CALL Z,FFFF
39	27	DAA	DAA
9	09	DAD B	ADD HL,BC
25	19	DAD D	ADD HL,DE
41	29	DAD HL	ADD HL,HL
57	39	DAD SP	ADD HL,SP
61	3D	DCR A	DEC A
5	05	DCR B	DEC B
13	0D	DCR C	DEC C
21	15	DCR D	DEC D
29	1D	DCR E	DEC E
37	25	DCR H	DEC H
45	2D	DCR L	DEC L
53	35	DCR M	DEC (HL)

Table C.2. (cont.)

Decimal	Hex	8085 Opcode	Z80 Opcode
11	0B	DCX B	DEC BC
27	1B	DCX D	DEC DE
43	2B	DCX H	DEC HL
59	3B	DCX SP	DEC SP
243	F3	DI	DI
251	FB	EI	EI
118	76	HLT	HALT
219	DB FF	IN FF	IN A, (FF)
60	3C	INR A	INC A
4	04	INR B	INC B
12	0C	INR C	INC C
20	14	INR D	INC D
28	1C	INR E	INC E
36	24	INR H	INC H
44	2C	INR L	INC L
52	34	INR M	INC (HL)
3	03	INX B	INC BC
19	13	INX D	INC DE
35	23	INX H	INC HL
51	33	INX SP	INC SP
218	DA FF FF	JC FFFF	JP C, FFFF
250	FA FF FF	JM FFFF	JP M, FFFF
195	C3 FF FF	JMP FFFF	JP FFFF
210	D2 FF FF	JNC FFFF	JP NC, FFFF
194	C2 FF FF	JNZ FFFF	JP NZ, FFFF
242	F2 FF FF	JP FFFF	JP P, FFFF
234	EA FF FF	JPE FFFF	JP PE, FFFF
226	E2 FF FF	JPO FFFF	JP PO, FFFF
202	CA FF FF	JZ FFFF	JP Z, FFFF
58	3A FF FF	LDA FFFF	LD A, (FFFF)
10	0A	LDAX B	LD A, (BC)

Table C.2. (cont.)

Decimal	Hex	8085 Opcode	Z80 Opcode
26	1A	LDAX D	LD A, (DE)
42	2A FF FF	LHLD FFFF	LD HL, (FFFF)
1	01 FF FF	LXI B, FFFF	LD BC, FFFF
17	11 FF FF	LXI D, FFFF	LD DE, FFFF
33	21 FF FF	LXI H, FFFF	LD HL, FFFF
49	31 FF FF	LXI SP, FFFF	LD SP, FFFF
127	7F	MOV A, A	LD A, A
120	78	MOV A, B	LD A, B
121	79	MOV A, C	LD A, C
122	7A	MOV A, D	LD A, D
123	7B	MOV A, E	LD A, E
124	7C	MOV A, H	LD A, H
125	7D	MOV A, L	LD A, L
126	7E	MOV A, M	LD A, (HL)
71	47	MOV B, A	LD B, A
64	40	MOV B, B	LD B, B
65	41	MOV B, C	LD B, C
66	42	MOV B, D	LD B, D
67	43	MOV B, E	LD B, E
68	44	MOV B, H	LD B, H
69	45	MOV B, L	LD B, L
70	46	MOV B, M	LD B, (HL)
79	4F	MOV C, A	LD C, A
72	48	MOV C, B	LD C, B
73	49	MOV C, C	LD C, C
74	4A	MOV C, D	LD C, D
75	4B	MOV C, E	LD C, E
76	4C	MOV C, H	LD C, H
77	4D	MOV C, L	LD C, L
78	4E	MOV C, M	LD C, (HL)
87	57	MOV D, A	LD D, A

Table C.2. (cont.)



Decimal	Hex	8085 Opcode	Z80 Opcode
80	50	MOV D,B	LD D,B
81	51	MOV D,C	LD D,C
82	52	MOV D,D	LD D,D
83	53	MOV D,E	LD D,E
84	54	MOV D,H	LD D,H
85	55	MOV D,L	LD D,L
86	56	MOV D,M	LD D,(HL)
95	5F	MOV E,A	LD E,A
88	58	MOV E,B	LD E,B
89	59	MOV E,C	LD E,C
90	5A	MOV E,D	LD E,D
91	5B	MOV E,E	LD E,E
92	5C	MOV E,H	LD E,H
93	5D	MOV E,L	LD E,L
94	5E	MOV E,M	LD E,(HL)
103	67	MOV H,A	LD H,A
96	60	MOV H,B	LD H,B
97	61	MOV H,C	LD H,C
98	62	MOV H,D	LD H,D
99	63	MOV H,E	LD H,E
100	64	MOV H,H	LD H,H
101	65	MOV H,L	LD H,L
102	66	MOV H,M	LD H,(HL)
111	6F	MOV L,A	LD L,A
104	68	MOV L,B	LD L,B
105	69	MOV L,C	LD L,C
106	6A	MOV L,D	LD L,D
107	6B	MOV L,E	LD L,E
108	6C	MOV L,H	LD L,H
109	6D	MOV L,L	LD L,L
110	6E	MOV L,M	LD L,(HL)

Table C.2. (cont.)

Decimal	Hex	8085 Opcode	Z80 Opcode
119	77	MOV M,A	LD (HL),A
112	70	MOV M,B	LD (HL),B
113	71	MOV M,C	LD (HL),C
114	72	MOV M,D	LD (HL),D
115	73	MOV M,E	LD (HL),E
116	74	MOV M,H	LD (HL),H
117	75	MOV M,L	LD (HL),L
62	3E FF	MVI A,FF	LD A,FF
6	06 FF	MVI B,FF	LD B,FF
14	0E FF	MVI C,FF	LD C,FF
22	16 FF	MVI D,FF	LD D,FF
30	1E FF	MVI E,FF	LD E,FF
38	26 FF	MVI H,FF	LD H,FF
46	2E FF	MVI L,FF	LD L,FF
54	36 FF	MVI M,FF	LD (HL),FF
0	00	NOP	NOP
183	B7	ORA A	OR A
176	B0	ORA B	OR B
177	B1	ORA C	OR C
178	B2	ORA D	OR D
179	B3	ORA E	OR E
180	B4	ORA H	OR H
181	B5	ORA L	OR L
182	B6	ORA M	OR (HL)
246	F6 FF	ORI FF	OR FF
211	D3 FF	OUT FF	OUT (FF),A
233	E9	PCHL	JP (HL)
193	C1	POP B	POP BC
209	D1	POP D	POP DE
225	E1	POP H	POP HL
241	F1	POP PSW	POP AF

Table C.2. (cont.)

Decimal	Hex	8085 Opcode	Z80 Opcode
197	C5	PUSH B	PUSH BC
213	D5	PUSH D	PUSH DE
229	E5	PUSH H	PUSH HL
245	F5	PUSH PSW	PUSH AF
23	17	RAL	RLA
31	1F	RAR	RRA
216	D8	RC	RET C
201	C9	RET	RET
32	20	RIM	-----
7	07	RLC	RLCA
248	F8	RM	RET M
208	D0	RNC	RET NC
192	C0	RNZ	RET NZ
240	F0	RP	RET P
232	E8	RPE	RET PE
224	E0	RPO	RET PO
15	0F	RRC	RRCA
199	C7	RST 0	RST 00
207	CF	RST 1	RST 08
215	D7	RST 2	RST 10
223	DF	RST 3	RST 18
231	E7	RST 4	RST 20
239	EF	RST 5	RST 28
247	F7	RST 6	RST 30
255	FF	RST 7	RST 38
200	C8	RZ	RET Z
159	9F	SBB A	SBC A,A
152	98	SBB B	SBC A,B
153	99	SBB C	SBC A,C
154	9A	SBB D	SBC A,D

Table C.2. (cont.)

Decimal	Hex	8085 Opcode	Z80 Opcode
155	9B	SBB E	SBC A,E
156	9C	SBB H	SBC A,H
157	9D	SBB L	SBC A,L
158	9E	SBB M	SBC A,(HL)
222	DE FF	SBI FF	SBC FF
34	22 FF FF	SHLD FFFF	LD (FFFF),HL
48	30	SIN	-----
249	F9	SPHL	LD SP,HL
50	32 FF FF	STA FFFF	LD (FFFF),A
2	02	STAX B	LD (BC),A
18	12	STAX D	LD (DE),A
55	37	STC	SCF
151	97	SUB A	SUB A,A
144	90	SUB B	SUB A,B
145	91	SUB C	SUB A,C
146	92	SUB D	SUB A,D
147	93	SUB E	SUB A,E
148	94	SUB H	SUB A,H
149	95	SUB L	SUB A,L
150	96	SUB M	SUB (HL)
214	D6 FF	SUI FF	SUB FF
235	EB	KCHG	EX DE,HL
175	AF	XRA A	XOR A
168	A8	XRA B	XOR B
169	A9	XRA C	XOR C
170	AA	XRA D	XOR D
171	AB	XRA E	XOR E
172	AC	XRA H	XOR H
173	AD	XRA L	XOR L
174	AE	XRA M	XOR (HL)
238	EE FF	XRI FF	XOR FF
227	E3	XTHL	EX (SP),HL

Table C.2. (cont.)

Decimal	Hex	8080 Opcode	Z80 Opcode
142	8E	ADC M	ADC A, (HL)
143	8F	ADC A	ADC A, A
136	88	ADC B	ADC A, B
137	89	ADC C	ADC A, C
138	8A	ADC D	ADC A, D
139	8B	ADC E	ADC A, E
206	CE FF	ACI FF	ADC A, FF
140	8C	ADC H	ADC A, H
141	8D	ADC L	ADC A, L
134	86	ADD M	ADD A, (HL)
135	87	ADD A	ADD A, A
128	80	ADD B	ADD A, B
129	81	ADD C	ADD A, C
130	82	ADD D	ADD A, D
131	83	ADD E	ADD A, E
198	C6 FF	ADI FE	ADD A, FF
132	84	ADD H	ADD A, H
133	85	ADD L	ADD A, L
9	09	DAD B	ADD HL, BC
25	19	DAD D	ADD HL, DE
41	29	DAD HL	ADD HL, HL
57	39	DAD SP	ADD HL, SP
166	A6	ANA M	AND (HL)
167	A7	ANA A	AND A
160	A0	ANA B	AND B
161	A1	ANA C	AND C
162	A2	ANA D	AND D
163	A3	ANA E	AND E
230	E6 FF	ANI FF	AND FF
164	A4	ANA H	AND H
165	A5	ANA L	AND L

Table C.3. Alphabetical list of Z80 opcodes

Decimal	Hex	8080 Opcode	Z80 Opcode
220	DC FF FF	CC FFFF	CALL C,FFFF
205	CD FF FF	CALL FFFF	CALL FFFF
252	FC FF FF	CM FFFF	CALL M,FFFF
212	D4 FF FF	CNC FFFF	CALL NC,FFFF
196	C4 FF FF	CNZ FFFF	CALL NZ,FFFF
244	F4 FF FF	CP FFFF	CALL P,FFFF
236	EC FF FF	CPE FFFF	CALL PE,FFFF
228	E4 FF FF	CPO FFFF	CALL PO,FFFF
204	CC FF FF	CZ FFFF	CALL Z,FFFF
63	3F	CMC	CCF
190	BE	CMP M	CP (HL)
191	BF	CMP A	CP A
184	B8	CMP B	CP B
185	B9	CMP C	CP C
186	BA	CMP D	CP D
187	BB	CMP E	CP E
254	FE FF	CPI FF	CP FF
188	BC	CMP H	CP H
189	BD	CMP L	CP L
47	2F	CMA	CPL
39	27	DAA	DAA
53	35	DCR M	DEC (HL)
61	3D	DCR A	DEC A
5	05	DCR B	DEC B
11	0B	DCX B	DEC BC
13	0D	DCR C	DEC C
21	15	DCR D	DEC D
27	1B	DCX D	DEC DE
29	1D	DCR E	DEC E
37	25	DCR H	DEC H
43	2B	DCX H	DEC HL
45	2D	DCR L	DEC L
59	3B	DCX SP	DEC SP
243	F3	DI	DI

Table C.3. (cont.)

Decimal	Hex	8080 Opcode	Z80 Opcode
251	FB	EI	EI
227	E3	XTHL	EX (SP),HL
235	EB	XCHG	EX DE,HL
118	76	HLT	HALT
219	DB FF	IN FF	IN A,(FF)
52	34	INR M	INC (HL)
60	3C	INR A	INC A
4	04	INR B	INC B
3	03	INX B	INC BC
12	0C	INR C	INC C
20	14	INR D	INC D
19	13	INX D	INC DE
28	1C	INR E	INC E
36	24	INR H	INC H
35	23	INX H	INC HL
44	2C	INR L	INC L
51	33	INX SP	INC SP
233	E9	PCHL	JP (HL)
218	DA FF FF	JC FFFF	JP C,FFFF
195	C3 FF FF	JMP FFFF	JP FFFF
250	FA FF FF	JM FFFF	JP M,FFFF
210	D2 FF FF	JNC FFFF	JP NC,FFFF
194	C2 FF FF	JNZ FFFF	JP NZ,FFFF
242	F2 FF FF	JP FFFF	JP P,FFFF
234	EA FF FF	JPE FFFF	JP PE,FFFF
226	E2 FF FF	JPO FFFF	JP PO,FFFF
202	CA FF FF	JZ FFFF	JP Z,FFFF
2	02	STAX B	LD (BC),A
18	12	STAX D	LD (DE),A
50	32 FF FF	STA FFFF	LD (FFFF),A
34	22 FF FF	SHLD FFFF	LD (FFFF),HL
119	77	MOV M,A	LD (HL),A
112	70	MOV M,B	LD (HL),B
113	71	MOV M,C	LD (HL),C
114	72	MOV M,D	LD (HL),D

Table C.3. (cont.)

Decimal	Hex	8080 Opcode	Z80 Opcode
115	73	MOV M,E	LD (HL),E
54	36 FF	MVI M,FF	LD (HL),FF
116	74	MOV M,H	LD (HL),H
117	75	MOV M,L	LD (HL),L
10	0A	LDAX B	LD A,(BC)
26	1A	LDAX D	LD A,(DE)
58	3A FF FF	LDA FFFF	LD A,(FFFF)
126	7E	MOV A,M	LD A,(HL)
127	7F	MOV A,A	LD A,A
120	78	MOV A,B	LD A,B
121	79	MOV A,C	LD A,C
122	7A	MOV A,D	LD A,D
123	7B	MOV A,E	LD A,E
62	3E FF	MVI A,FF	LD A,FF
124	7C	MOV A,H	LD A,H
125	7D	MOV A,L	LD A,L
70	46	MOV B,M	LD B,(HL)
71	47	MOV B,A	LD B,A
64	40	MOV B,B	LD B,B
65	41	MOV B,C	LD B,C
66	42	MOV B,D	LD B,D
67	43	MOV B,E	LD B,E
6	06 FF	MVI B,FF	LD B,FF
68	44	MOV B,H	LD B,H
69	45	MOV B,L	LD B,L
1	01 FF FF	LXI B,FFFF	LD BC,FFFF
78	4E	MOV C,M	LD C,(HL)
79	4F	MOV C,A	LD C,A
72	48	MOV C,B	LD C,B
73	49	MOV C,C	LD C,C
74	4A	MOV C,D	LD C,D
75	4B	MOV C,E	LD C,E
14	0E FF	MVI C,FF	LD C,FF
76	4C	MOV C,H	LD C,H

Table C.3. (cont.)



Decimal	Hex	8080 Opcode	Z80 Opcode
77	4D	MOV C,L	LD C,L
86	56	MOV D,M	LD D,(HL)
87	57	MOV D,A	LD D,A
80	50	MOV D,B	LD D,B
81	51	MOV D,C	LD D,C
82	52	MOV D,D	LD D,D
83	53	MOV D,E	LD D,E
22	16 FF	MVI D,FF	LD D,FF
84	54	MOV D,H	LD D,H
85	55	MOV D,L	LD D,L
17	11 FF FF	LXI D,FFFF	LD DE,FFFF
94	5E	MOV E,M	LD E,(HL)
95	5F	MOV E,A	LD E,A
88	58	MOV E,B	LD E,B
89	59	MOV E,C	LD E,C
90	5A	MOV E,D	LD E,D
91	5B	MOV E,E	LD E,E
30	1E FF	MVI E,FF	LD E,FF
92	5C	MOV E,H	LD E,H
93	5D	MOV E,L	LD E,L
102	66	MOV H,M	LD H,(HL)
103	67	MOV H,A	LD H,A
96	60	MOV H,B	LD H,B
97	61	MOV H,C	LD H,C
98	62	MOV H,D	LD H,D
99	63	MOV H,E	LD H,E
38	26 FF	MVI H,FF	LD H,FF
100	64	MOV H,H	LD H,H
101	65	MOV H,L	LD H,L
42	2A FF FF	LHLD FFFF	LD HL,(FFFF)
33	21 FF FF	LXI H,FFFF	LD HL,FFFF
110	6E	MOV L,M	LD L,(HL)
111	6F	MOV L,A	LD L,A
104	68	MOV L,B	LD L,B

Table C.3. (cont.)

Decimal	Hex	8080 Opcode	Z80 Opcode
105	69	MOV L,C	LD L,C
106	6A	MOV L,D	LD L,D
107	6B	MOV L,E	LD L,E
46	2E FF	MVI L,FF	LD L,FF
108	6C	MOV L,H	LD L,H
109	6D	MOV L,L	LD L,L
49	31 FF FF	LXI SP,FFFF	LD SP,FFFF
249	F9	SPHL	LD SP,HL
0	00	NOP	NOP
182	B6	ORA H	OR (HL)
183	B7	ORA A	OR A
176	B0	ORA B	OR B
177	B1	ORA C	OR C
178	B2	ORA D	OR D
179	B3	ORA E	OR E
246	F6 FF	ORI FF	OR FF
180	B4	ORA H	OR H
181	B5	ORA L	OR L
211	D3 FF	OUT FF	OUT (FF),A
241	F1	POP PSW	POP AF
193	C1	POP B	POP BC
209	D1	POP D	POP DE
225	E1	POP H	POP HL
245	F5	PUSH PSW	PUSH AF
197	C5	PUSH B	PUSH BC
213	D5	PUSH D	PUSH DE
229	E5	PUSH H	PUSH HL
201	C9	RET	RET
216	D8	RC	RET C
248	F8	RM	RET M
208	D0	RNC	RET NC
192	C0	RNZ	RET NZ
240	F0	RP	RET P
232	E8	RPE	RET PE
224	E0	RPO	RET PO

Table C.3. (cont.)

Decimal	Hex	8080 Opcode	Z80 Opcode
200	C8	RZ	RET Z
23	17	RAL	RLA
7	07	RLC	RLCA
31	1F	RAR	RRA
15	0F	RRC	RRCA
199	C7	RST 0	RST 00
207	CF	RST 1	RST 08
215	D7	RST 2	RST 10
223	DF	RST 3	RST 18
231	E7	RST 4	RST 20
239	EF	RST 5	RST 28
247	F7	RST 6	RST 30
255	FF	RST 7	RST 38
158	9E	SBB M	SBC A, (HL)
159	9F	SBB A	SBC A, A
152	98	SBB B	SBC A, B
153	99	SBB C	SBC A, C
154	9A	SBB D	SBC A, D
155	9B	SBB E	SBC A, E
156	9C	SBB H	SBC A, H
157	9D	SBB L	SBC A, L
222	DE FF	SBI FF	SBC FF
55	37	STC	SCF
150	96	SUB M	SUB (HL)
151	97	SUB A	SUB A, A
144	90	SUB B	SUB A, B
145	91	SUB C	SUB A, C
146	92	SUB D	SUB A, D
147	93	SUB E	SUB A, E
148	94	SUB H	SUB A, H
149	95	SUB L	SUB A, L
214	D6 FF	SUI FF	SUB FF
174	AE	XRA M	XOR (HL)
175	AF	XRA A	XOR A

Table C.3. (cont.)

Decimal	Hex	8080 Opcode	Z80 Opcode
168	A8	XRA B	XOR B
169	A9	XRA C	XOR C
170	AA	XRA D	XOR D
171	AB	XRA E	XOR E
238	EE FF	XRI FF	XOR FF
172	AC	XRA H	XOR H
173	AD	XRA L	XOR L

Table C.3. (cont.)



## Appendix D Bibliography

MCS-80/85 Family User's Manual, Intel Corporation, 3065 Bowers Avenue, Santa Clara, CA 95051. Order no. 205775. \$7.50.

8080 8085 Assembly Language Programming Manual, Intel Corporation, 3065 Bowers Avenue, Santa Clara, CA 95051. Order no. 980940. \$13.50.

8080/8085 Software Design, Book 1, by Christopher A. Titus, David G. Larsen, and Jonathon A. Titus, Howard W. Sams & Co., Inc. 4300 West 62nd St. Indianapolis, Indiana 46268, 1978. ISBN 0-672-21541-1. \$12.95.

8080/8085 Software Design, Book 2, by Christopher A. Titus, David G. Larsen, and Jonathon A. Titus, Howard W. Sams & Co., Inc. 4300 West 62nd St. Indianapolis, Indiana 46268, 1978. ISBN 0-672-21615-9. \$12.95.

8080A-8085 Assembly Language Programming by Lance A. Leventhal, Osborne/McGraw-Hill, Berkeley, California, 1978, ISBN 0-931988-10-1. \$18.95

8080 8085 Assembly Language Subroutines by Lance A. Leventhal, Winthrop Saville, Osborne/McGraw-Hill, 2600 Tenth Street, Berkeley, California 94710, 1983, ISBN 0-931988-58-6. \$17.95.



## Memory Index

Address	Page No.	Address	Page No.
0009	45	190F	194
001E	228	192F	194
0020	226	1962	194
0024	241, 245, 253	1996	277
002C	234, 241, 245	1999	276
0034	129, 241, 245	19A0	195
003C	241, 245	1B32	241
0040-0041	262	1B35	241
035A-036C	277	1B3B	243
036F-03E2	262, 277	1BE0	228
051D	103	1E5E	183
0858	44	1FBE	275
08DB	260	220F	274
0C5F	103	22B9	211
0FE8	276	2413	211
1069	44	2542	276
113B	226	260B-2656	212
1140	226	27B1	228
11A2	228	290F	276
12CB	102	2C34	50
13DB	102	2E40	50
1431-1458	57, 241	2EE6	276
1470	183	33DC	44
14A8	210	3469	276
14AA	210	3472	276
14B0-14C0	211	4222	225
14C1	210	4225	223
14D8-17DB	258	4229	223
17E6	140	4229	225
422D	223	5A9E	105
4231	223	5AA9	275



Address	Page No.	Address	Page No.
4231	225	5AE3	275
4235-426E	224	5B3E	105
427C	227	5B46	105
428A	105	5B68-5B74	71
42A5	105	5B79	105
42A8	105	5BD2	103
438A-43A1	223	5D0A-5D2B	105
43AF	224	5D64	103
43B2	223	5E15	105
43B8-43F9	223	64192	63
4431-4439	224	65C3	276
443B	105	6AC3	57
443F-444A	224	6B61-6B9F	275
4453-44AA	223	6BDB	276
44AF-4537	224	6BE6	276
4548	223	6CA9	277
4548	223	6CD6	272
4548	224	6CE0	271
454E	224	6CE4	243
457D	243	6CE5	142
4584	243	6D3F-6D6C	179, 182
4644	103	6D69	243
4B3F	226	6D6D	141
4B44	44, 222, 226	6D7E	141
4B55	183	6DAC	129
4BA0	184	6DAC	241
4D59	258	6DE6-6F30	162
506B	260	6E0B	141
5071-5073	259	6E1E	142
516A	103	6E32	142
51A4	105	6E75-6EA6	140
52BB	162	6EAA-6EB8	132
52D0	162	6ECB	141
532D	162	6EE5-6EED	162
5443	105	6EEA	170
54BC	226	6EEF	161
55CD	258	6EF2	162
55D4	103	6F2C	162
5791	228	6F46	210
5797	271	6F5B	210

Address	Page No.	Address	Page No.
5A58	228	6F5B-6F84	206
5A62	276	6F85-7042	211
5A7C	104	6FDB-7015	205
702A	211	F602	241
718E-71B2	107	F605-FC98	277
71F6	243	F62A	262
720D	107	F62B	162
7233-7241	108	F639-F63E	222, 227
7242	101	F63D	105
726D	243	F648	222
7270	102	F65B-F65F	141
7283	102	F674	183
729F	205	F675	222, 226, 228
72C5	172	F685	103
72C5-7303	171	F788	222
7304	259	F923-F92E	195
7326	259	F962-FA74	272
7329-7390	195, 196	F9BD-F9C2	262
7383-7390	190, 191	FAAE	84, 86, 161
73EA	243	FABE	254
73EE	218	FADA-FB39	44, 260
740F	210	FB0C-FB12	260
743E	243	FB1A-FB26	258
744C-744D	226	FB28	259
7657-767C	169	FB28-FB29	260
765C	242, 243	FB2A	259
7662	168	FB2C	260
7662	223	FB2E-FB30	259
7676-767C	168, 205	FB39	44
767D-770A	259	FBB2	273
7711-7BF0	218	FC93-FC98	274
78F1	218	FCC0	222
7BF1-7CF8	105	FDFF	222
7CF9-7D06	108	FE00-FF40	217, 222, 279
7D07	108	FF40-FFFD	222, 272
7D0B-7D10	107	FF41	142
7D1B-7D2F	108	FF42	140, 141, 142
7D33	44	FF44	162, 170, 205
7D44	99	FF45	84, 191, 208, 260
7D46-7D4C	100	FF8B	140
7FD6	44, 45		

<b>Address</b>	<b>Page No.</b>
7FF3	45, 260
E000-E0FF	272
F5F0-F602	254, 277
F5F9	234, 241
F5FC	129, 241, 244
F5FF	192, 241

# Alphabetical Index

## A

A register	31
AC register	46
AC adaptor	254
AC flag	32
accumulator	31
ACI opcode	47
acoustic coupler	157
ACP/DIR switch	88, 133
ADC opcode	47
ADD opcode	46
add-with-carry	47
address decoder	82
ADI opcode	46
AF register	31
ALC	208
ANA opcode	52
AND opcode	51, 52
ANI opcode	52
ANS/ORIG switch	88
answer mode	147
arrow key	108
ASCII code	130, 281
assembler	26
assertion, RS-232	136
asynchronous	120
automatic level control	208
auxiliary carry flag	32, 46

## B

B register	31
BA files	273
backup power	248
Bar Code Reader	231
baud rate	118, 121, 279
Baudot code	130
Baudot, J.M.E.	121
BAUDST routine	140
BC register	31, 33
BCD code	46, 49

BCR signal	82, 87, 231, 245
BCR hook	277
BEEP command	132, 163, 225
beeper	205
BEGLCD routine	222
Bell 103	144
bit	25
bit time	120
block moves	276
BRKCHK routine	102
Buck, Alan	19
buffer, keyboard	101, 103
bus, parallel	117
BUSY signal	87
BUSYNOT signal	87
byte	25

## C

C flag	46
C register	31
calendar	187
CALL	41, 42, 68
CARDET routine	161
carry flag	31, 32
CASIN routine	211
Cassette	199
CC opcode	42
CCF opcode	56
CCITT standard	148
CD signal	133
Centronics standard	175
character length selection	123
CHGET routine	102
CHKDC routine	275
CHSNS routine	102
CL/AS signal	133
CLEAR command	64
CLOADM command	67, 200
clock	187

CLRFNK routine	105	CSOUT routine	210
CLS routine	225	CSRX, CSRY routine	222
CLS1, CLS2 signal	123	CTOFF, CTON	
CLSCOM routine	141	routine	210
CM opcode	42	CTS signal	87, 133, 137
CMA opcode	56	CTSR signal	133
CMC opcode	56	CUROFF routine	224
CMOS	79	current loop	147
CMP opcode	53	CURSON routine	224
CMT device	201	cursor keys	108
CN1-9	89	CY flag	32, 46
CN2	236	cycles	33
CN3	202	CZ opcode	43
CN4	266		
CN7	84	<b>D</b>	
CN9	254	D register	31
CNC opcode	42	DAA opcode	46, 50
CNZ opcode	43	DAD opcode	50
CO files	68, 274	data received	126
CODE keys	105	DATAIN,	
command processing	277	DATAOUT signal	87
comparison	53	DATAR routine	211
complement	56	DATAW routine	210
complement carry flag	57	DATE, DAY routine	194
conditional calls	42	DB connectors	135
conditional execution	41	DCR opcode	49
CONN routine	162	DCX opcode	50, 51
connectors	89	DE register	31, 33
CP opcode	42, 53	DEC opcode	49, 51
CP/TL signal	133	decimal	25
CPE opcode	42	decoder, address	82
CPI opcode	53	decrement	49, 50
CPL opcode	56	DELLIN routine	224
CPO opcode	43	denial, RS-232	136
CPU chip	22, 81	dextrose	216
CRL signal	124	DI opcode	57
CRLF routine	225	DIAL routine	161, 162
cross-assembler	77, 79	DIR/ACP switch	144, 157
CRT device	257	direct addressing	37
crystals, generally	90	direct-connect	149
crystal X1	188	directory	278
crystal X2	33, 125	disabling interrupts	242
CSAVEM command	66, 200	DISC routine	162

discharge memory 248  
 disk I/O 259  
 DISP control 217  
 DO files 273  
 DR signal 82, 126, 245  
 DSPFNK routine 105  
 DSR signal 87, 133, 137  
 DSRR signal 133  
 DTMF 148  
 DTR signal 88, 133, 137  
 DTRR signal 87

## E

E register 31  
 EBCDIC code 130  
 EI opcode 57  
 EIA standard 135  
 8080 18, 33, 52, 73  
 8080 mnemonic 33  
 81C55 86  
 electret condenser microphone 159  
 Electronic Industries Ass'n 135  
 end address, endadd 66, 200  
 ENDLCD routine 222  
 ENTREV routine 224  
 entry address 66  
 EPE signal 123  
 ERAEOL routine 224  
 ERAFNK routine 105  
 ESCA routine 225  
 Escape sequences, LCD 223  
 Escape-Y sequence 227  
 EX (SP),HL opcode 40  
 EX DE,HL opcode 39  
 expansion bus 262  
 expression 60  
 EXTREV routine 224

## F

F register 31, 32  
 f-string 104  
 F6, F7 Telcom key 260  
 FC error 45  
 FCC certification 152  
 FE signal, error 129  
 Find 21  
 flag register 31  
 FNKSB routine 105  
 framing error 129  
 FRE function 64  
 frequency-shift keying 144  
 FSK 144  
 function 60

## G

gaps 74  
 GOTO command 24  
 GRPH keys 105  
 GTXTTB routine 275

## H

H register 31  
 HALT opcode 57  
 handshaking 135, 137  
 Hayashi 273  
 hexadecimal 25  
 hexadecimal conversion 276  
 high-order part 33, 62  
 HIMEM function 64, 277  
 HL register 31, 33  
 HLT opcode 57  
 HOME routine 223  
 hookswitch 151

## I

I, O ports 59, 86  
 IE opcode 246

IM6402 chip	121
immediate move	36
IN opcode	37, 59
INC opcode	49, 51
increment	49, 50
indirect addressing	36
INITIO routine	272
INKEY\$ function	101
INLIN routine	103
INP function	59, 62
input ports	62
INR opcode	49
INSCHR routine	275
INSLIN routine	224
interrupts	239
INTR signal	82, 245
INX opcode	50, 51
INZCOM routine	140
IOINIT routine	271

**J**

jamming	43
JC opcode	41
JM opcode	41
JMP opcode	40, 41
JNC, JNZ, JP opcode	41
JP (HL) opcode	45
JPE, JPO opcode	41
jump	24
JZ opcode	41

**K**

KBCHAR routine	102
KBLINE routine	103
keyboard	93
keyboard buffer	103
keyboard scanning	96
KEYX routine	102
KILASC routine	275
Kreindler, Lee	19
KYREAD routine	101

**L**

L register	31
label line	103
last-in first-out	33
LCD routine	215, 222
LCOPY command	183, 217
LD opcode	33, 36, 37
LDA, LDAX opcode	37
leader, tape	208
LFILES command	260
LHLD opcode	38
LIFO	33
Liljedahl, Harold	19
LINE command	218
LIST, LLIST	
command	226
load opcode	33, 36
LOADM command	67, 200
LOC function	259
LOCK routine	224
LOF function	259
Low Battery lamp	184, 252
low-order portion	33, 62
low-power signal	130, 252
lowercase conversion	276
LPOS function	182, 278
LPS signal	240, 245
LPS hook	277
LXI opcode	38

**M**

M11 chip	80
M12 chip	80
M14 chip	84
M15 chip	84
M18 chip	188
M22 chip	84, 121
M23 chip	84
M24 chip	84
M25 chip	84
M35 chip	140

M36 chip	84
MAKHOL routine	275
MAKTXT routine	274
marking	147
MASDEL routine	275
masking interrupts	242
MAXRAM function	63
MC14412 chip	155
memory map	27, 81
memory power	248
MENU routine	271
menu selection	68
microphone	159
mnemonics	33, 74
Model I, III, IV	18
modem	117, 143
motor control	208
MOV opcode	33, 36
move opcode	33, 36
multiplexer	118, 132, 144
MUSIC routine	171
MVI opcode	37
<b>N</b>	
nicad, nickel-cadmium	248
no-op, NOP opcode	57
null modem	137
NUMS keys	108
<b>O</b>	
OE signal, error	129
off-hook	148
ON ... GOSUB command	244
on-hook	148
ON...GOTO command	44, 45
opcodes	24, 74
operating system	24
option ROM	80, 278
optocoupler	236
OR opcode	51, 52
ORA opcode	52
ORI opcode	52
ORIG/ANS switch	133, 144, 153
originate	147
OT1	144, 149, 157
OT2	248
OUT opcode	37, 59, 63
output port	63
overflow error	129
<b>P</b>	
P flag	32
parallel bus	117, 118
parity error	129
parity flag	32
parity, UART	123
Paste buffer	278
PC register	33
PCHL opcode	45
PCS signal	87, 241, 253
PE signal, error	129
PEEK function	60, 274
PI signal	123
Piezoelectric effect	163
PIO chip	86
PIO timer	125, 171
pixel	215
PLOT routine	218, 226
PNOTAB routine	183
POKE command	60, 61, 274
Polaroid	215
polling	234
POP opcode	39
port map	85
ports, I/O	86
POS column	278
POSIT routine	227
power control	132
power supply	247
pps flag	278
PRESET command	217
Printer	175
PRINTR routine	182
program counter	33



PRTLCD routine	183	RPE, RPO opcode	43
PRTTAB routine	183	RR register	121
PSET command	217	RRA, RRC opcode	54, 87
pseudo-ASCII	100	RRC signal	121
PSW	40	RRCA opcode	54
PUSH opcode	39	RRI signal	121, 133
<b>R</b>		RS 232 port	117
RAL, RAR opcode	54	RS 232C standard	135
RBR	121	RS232C signal	87
RC opcode	43	RST opcode	43, 44, 45
RCVX routine	141	RST 3 opcode	277
receiver	118	RST 4 opcode	183, 226
receiver buffer		RST 5.5 opcode	82, 231, 245
register	121	RST 6.5 opcode	82, 126, 245
receiver register	121	RST 7.5 opcode	82, 192, 245
receiver register clock	121	RSTSYS routine	224
receiver register input	121	RTS signal	88, 133, 137
register indirect		RTSM signal	133
addressing	37	RTSR signal	133
registers	31, 33	RV232C routine	141
relative jumps	70	RXC signal	202
relays	90	RXCAR signal	133, 153
REM1, REM2 signal	210	RXD, RXDB signal	231
REMOTE signal	89	RXM signal	133
REN	149	RXMD, RXMe signal	151
reset	31, 44, 253	RXMi signal	153
RESET*	253	RXR signal	133
restart	43	RY1	208
RET opcode	42, 43	RY2	149, 151
return	42	RY3	151
RIM opcode	57, 205, 234	RZ opcode	43
ring pulse	129, 266	<b>S</b>	
ring signal	148	S flag	32
ringer equivalence		SAVEM command	66
number	149	SBB opcode	48
RLA, RLC, RLCA		SBC opcode	48
opcode	54	SBI opcode	48
RM, RNC, RNZ		SBS signal	123
opcode	43	SCF opcode	56
Robbie, Gerald	19	SD232C routine	142
rotate	54	SENDCCQ routine	141
RP signal	43, 88, 129, 151	SENDCCS routine	142

serial bus	117	SW2	157
set	31	SW3	252
set carry flag	57	SW4	90
SETSER routine	140	SW5	90
SETSYS routine	224	switches	90
shift	76	SYNCR routine	211
SHIFT keys	105	SYNCW routine	210
SHLD opcode	38		
SID signal	56, 82, 201, 246	<b>T</b>	
sign flag	32	TBR	118
SIM opcode	57, 205, 234	TBRE signal	126
sixteen-bit arithmetic	50	TELCOM	21
sixteen-bit transfers	38	teletype	147
SNDCOM *6E3A		TEXT	21
SOD signal	82, 201, 245	TIME routine	194
SOUND command	163, 172, 279	tip signal	148
SP register	31, 33	TL	151
space	147	Touch-Tone	148
SPHL opcode	40	TP signal	82, 192, 241, 245
STA opcode	37	TP hook	277
stack operations	39	TR	118
stack pointer	33	transfer address	66
stadd	200	transmitter	118
start address	66	transmitter buffer	
start bit	120	register	118
Stat setting	141, 278	transmitter buffer	
STAX opcode	37	register empty	126
STB signal	89	transmitter receiver	
STC opcode	56	clock	121
STDSPF routine	105	transmitter register	118
STFNK routine	104	transmitter register	
stop bit	120	output	121
stop bit select	123	TRAP signal	240, 245, 253
STROBE signal	89	TRC signal	87, 121
strobe, printer	177	TRO signal	121, 133
STROM signal	89	TRS-80 Model I, III,	
SUB opcode	48	IV	18
subroutines	41	TS	87
subtract	48	turn bit off, on	31, 52, 53
subtract immediate	48	TX signal	157
SUI opcode	48	TXC signal	203
Suzuki	273	TXM signal	133
SW1	90	TXMD signal	151

TXMe signal	151
TXMi signal	157
TXR signal	133

## U

UART	117
UART DR hook	277
UART interrupt	241
universal	120
UNLOCK routine	224
UNPLOT routine	218, 226
unused pins	259

## V

VARPTR function	68
VB power	188
VDCHAR routine	222
VDCLS routine	225

## W

WAND device	257
word	118

## X

X1 crystal	188
X2 crystal	125
XCHG opcode	39
XON/XOFF	130, 132, 140, 279
XOR opcode	51, 52
XRA opcode	52
XRI opcode	52
XTHL opcode	40

## Y

Y0-Y7 signals	59, 62, 86
---------------	------------

## Z

Z flag	32
Z80	18, 33, 73
Z80 mnemonics	33
zero flag	32

\$19.95

## Inside the TRS-80 Model 100

*Inside the TRS-80 Model 100 Computer* is a thorough guide to the internal workings of the Radio Shack Model 100, the best-selling laptop computer. This book is a valuable source of information for those who wish to learn machine-language programming and provides powerful new techniques for BASIC programmers. Topics discussed include:

- disassembly of critical ROM routines
- keyboard scanning
- UART, RS-232 and modem
- clock/calendar chip
- cassette and RAM file structure
- interrupt handling
- 8085 instruction set
- explanation of SAVEM, LOADM, CALL
- explanation of INP, OUT, PEEK, and POKE
- I/O port map
- power supplies

Commented ROM routine listings, detailed hardware schematics, and numerous figures are provided throughout the book. *Inside the TRS-80 Model 100* also contains invaluable appendices and is fully indexed. No Model 100 owner, whether novice or expert, should be without this book.

ISBN: 0-938862-31-6

LC: 85-11560



WEBER SYSTEMS, INC.

8437 Mayfield Road, Chesterland, Ohio 44026